Virtual Integration
A Game-Theoretic Approach

Inaugural-Dissertation zur Erlangung der Doktorwürde
der Fakultät für Sprach-, Literatur- und
Kulturwissenschaften der Universität Regensburg

vorgelegt von
Michael Schorer aus
Kempten (Allgäu)

Die Arbeit wurde im Jahr 2014 von der
Fakultät für Sprach-, Literatur- und Kulturwissenschaften
der Universität Regensburg als Dissertation angenommen.

Erstgutachter: Prof. Dr. Christian Wolff

Zweitgutachter: Prof. Dr. Rainer Hammwöhner

Drittgutachter: Prof. Dr. Jürgen Mottok

Abstract

Embedded systems are playing a key role to enable the features of todays cars and other road vehicles. Advances in current hardware platforms of these embedded systems and growing implementation of features in software rather than in new hardware modules lead to new and continuously more complex automotive software systems.

The integration process in the course of development of automotive software systems is a crucial and complex phase. An efficient combination of the set of components into a functioning whole is made difficult due to the fine granular and highly interconnected architectural structure of these systems. Also, the high cost pressure and strict safety requirements in the industry have strong impact on the integration phase.

Virtual integration is a proposed methodology, which aims to carry out integration-related activities at an early stage of development. This reduces the pressure during the integration phase and improves the quality of the development process and the resulting software product.

This thesis presents the requirements of a suitable tool chain for the virtual integration and the corresponding reference architecture. The resulting compound information system addresses the following virtual integration subprocesses: Compatibility verification, integration planning, integration monitoring, integration administration and build automation.

The work further features a feasibility study of interface automata for the compatibility verification subprocess of the virtual methodology. Interface automata present a light-weight mechanism to capture interface behavior of software components and incorporate a convenient verification technique, which is based on game-theoretic foundations.

Integration planning is the process of coordination of the integration activities over the development time. A set of integration measurement techniques present a method to measure quality of such integration plans. Integration games are presented as a novel optimization method for integration planning. They capture essential aspects of integration through the succinct game representation and enable optimization by established game solving algorithms.

Contents

List of Figures

List of Tables

List of Abbreviations

AGG	Action Graph Game
AGGFN	Action Graph Games with Function Nodes
BR	Build Rules
CDO	Connected Data Objects
COTS	Component off-the-shelf or Commercial off-the-shelf
CR	Compatibility Check Result
CUT	Component under Test
DCC	Dynamic Compatibility Checker
DSS	Decision Support System
ECU	Engine Control Unit
EMF	Eclipe Modeling Framework
EMOF	Essential MOF
HIL	Hardware in the Loop
IA	Interface Automaton
IGM	Integration Game Model
IIS	Integration Information System
IP	Integration Plan
IPS	Integration Planning System
IT	Integration Trace
KS	Knowledge System
LS	Language System

M2M	Model to Model
MOF	Meta Object Facility
OCL	Object Constraint Language
OMG	Object Management Group
PP	Project Plan
PPS	Problem-Processing System
PS	Presentation System
QVT	Queries, Views, Transformations
ROM	Read Only Memory
RP	Release Plan
SA	System Architecture
SB	Software Build
SC	Source Code
SI	System under Integration
SIL	Software in the Loop
TIA	Timed Interface Automaton
TR	Test Report
VitaS^3	Virtual and Automated Integration of Software Functionalities in Distributed Embedded Automotive Systems with regards to safety requirements.
XMI	XML Metadata Interchange
XML	Extensible Markup Language
XSD	XML Schema Definition

Chapter 1

Introduction

SOFTWARE development for embedded systems in the automotive domain is characterized through high cost pressure, restrictive safety requirements and real-time requirements, a high grade of componentization, software reuse and also the integration of legacy third party and customer components. The combination of these characteristics implies various challenges during the development of embedded software systems.

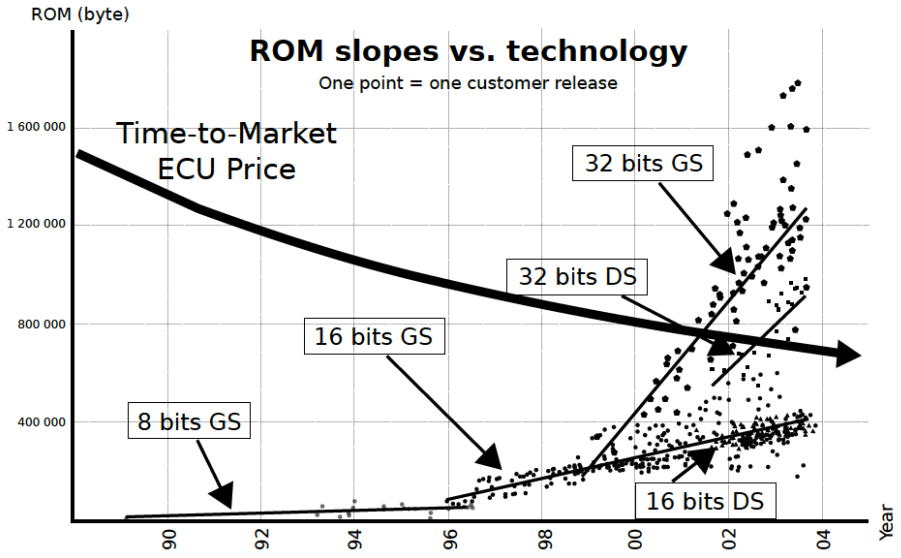


Figure 1.1: Development of engine management system technology (cf. Claraz, Eppinger, and Berentroth, 2004b). The change to 32-Bit systems increased the demand for ROM in the ECU (Embedded Control Unit) for both gasoline (GS) and diesel (DS) systems. The time-to-market and the ECU price has been steadily decreasing in the time frame.

The trend of the number of electronic control systems, as depicted in figure 1.1, shows that more and more capable hardware is deployed to support the complexity of the software solutions, embedded in todays automotive systems. This means that challenges are about to increase even further in the future.

Future infotainment and comfort solutions, safety solutions like drive-by-wire or break-by-wire systems and especially the shift from combustion engine systems towards hybrid and fully electric engine concepts will intensify the current challenges.

About 90% of all innovations in the automotive sector are happening through software BITKOM, 2010; IBM, IDC, Mercer, and M. Analysis, 2010. This leads to an increasing number of ECUs and software functions. A typical engine management system, for example, has about 1 million lines of code, 5000 individual software functions. These functions exchange about 20000 different signals that lead to about 50000 dependency relations between functions.

The development of these is carried out in an organizational framework with developers spread over different locations and time zones. Together with the tight project schedules that impose strict deadlines for the delivery of customer releases, this leads to a high complexity in the integration and demands a high grade of coordination of the integration tasks.

In Hiemann, 1975 the correlation between the point in time when an error is detected during the software development process and the cost for its removal is described. The later an error is discovered in the software development process, the higher is the cost for error correction. Since the integration phase is located at a late point in the process, the bug fixing costs for errors in the integration phase are very high and the need for error removal has a big impact on overall project costs.

In terms of functionality, automotive manufacturers and the automotive supplier industry have established a profound knowledge and the skills to implement software systems that meet the strict quality and safety requirements. They also implemented a set of supportive processes along the software development methods.

Figure 1.2 shows the iterative v-model and the location of support processes and tools. For the requirement analysis and definition phase the automotive companies are follow internal processes and use requirement management solutions like Rational DOORS. The system architecture phase is supported through the use of architecture description languages like EAST-ADL, EAST-ADL2 or SAE-AADL. Furthermore, the use of standardized architectures for the automotive domain like AUTOSAR provides developers with processes and tools that support the design phase of the software development process.

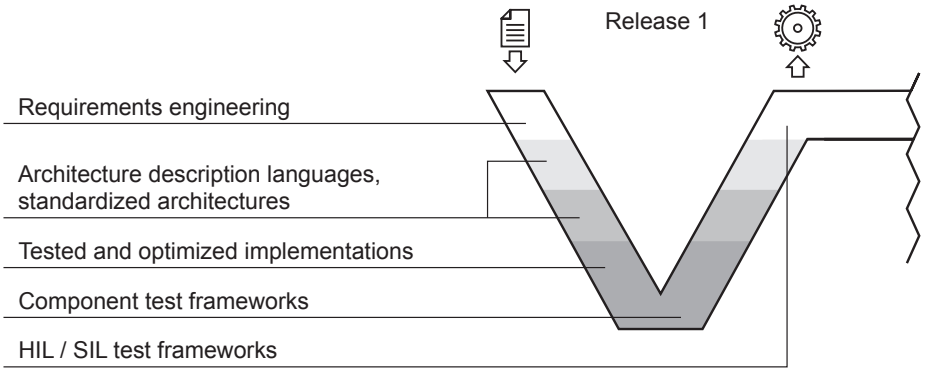


Figure 1.2: Tooling and development support during a cycle of the V-model.

Furthermore, there are several verification & validation frameworks that are used throughout the design phases, e.g. real-time analysis from tool vendors like Timing Architects or Symptavision.

Software developers have gathered a rich set of implementations from previously successfully completed projects that can be reused to satisfy the functional requirements of future products. The software components' core functionalities have been tested thoroughly and have been optimized for performance and an efficient use of the hardware platform.

Software components that are designed and implemented from scratch go components, which are altered significantly from their previous implementation, are tested intensively with use of test frameworks. Hardware in the Loop (HIL) and Software in the Loop (SIL) testing is used mainly in the system testing phase to test the software components' functionality in its environment.

1.1 Research Project VitaS^3

The application and following funding of the research project VitaS^3 was a result of the challenges that integrators are facing in the industry. The research project has the title: *Virtual and Automated Integration of Software Functionalities in Distributed Embedded Automotive Systems with regards to safety requirements.* The project had two industry partners with Continental Automotive GmbH and iNTENCE automotive electronics GmbH. There were three academic partners part of VitaS^3: the University of Applied Sciences Regensburg and the LaS^3, the Faculty for Language,

Literature and Cultural Studies at the University of Regensburg and the University of West Bohemia.

Figure 1.2 illustrated the wide range of tooling and development support methods that are currently used in software development projects in the automotive industry. The illustration shows that the integration phase does not have any kind of development support. The goal of VitaS^3 is to examine the reasons for this blank spot and to propose adequate processes and tooling to improve the software integration phase.

VitaS^3 consists of ten work packages:

1. Compatibility and Variability as Factor of Influence for Integration
2. Analysis of Integration and Integration Test Strategies (State of the Art Analysis)
3. Interface Test
4. Interface Automata
5. Petri Nets in the (Virtual) Integration
6. Modeling Integration Aspects with the Synchronous Languages Esterel and Lustre
7. Task Modeling and Configuration in the Virtual Integration
8. Analysis of the existing Software Architecture of at least one Specific Embedded Automotive System
9. Application and Verification of the Concepts of the scientific Work Packages (Demonstrator WP)
10. Transfer Analysis and Product Development

The research project is funded by the Bavarian State Ministry of Sciences, Research and the Arts over a timespan of 3 years with a funding of 259.742 Euro. VitaS^3 employed two full-time PhD candidates and four student assistants (internships and diploma theses).

1.1.1 Industry Context

In the course of the research project, a workshop was held with industry expert form iNTENCE automotive electronics. The experts at iNTENCE automotive electronics are working in projects at various automotive suppliers and also directly in automotive OEMs. The workshop included a presentation of the research project and an exchange of the current software integration situation in the industry. The results of this workshop are presented in the following. The focus of this workshop was

to provide a common understanding and terminology of the activities during the integration phase, to get a state-of-the-art overview in the involved industry partners project settings and to identify the challenges that integrators are facing. The workshop also offered the possibility to receive suggestions and recommendations that help to perform a successful integration from the industry experts.

Integration Experience Integration projects at iNTENCE are mainly small-scale controller implementations like airbag controllers, pedestrian safety or trailer controllers. The number of components is smaller than 50 in most cases. These projects are mostly not consistently model-based, i.e., there are models only for certain components, e.g., functional modeling with tools like Matlab Simulink or Stateflow of critical core components, or for certain aspects, e.g., requirements model from tools like Rhapsody.

The most common integration strategy for new products is the bottom-up integration strategy (see section 2.5.2 for details), i.e., the integration starts with the core components, which implement the core functionality of the product. The bottom-up integration strategy is often aligned to the project schedule, which leads to an integration strategy that is influenced by system architecture as well as project specific constraints. Integrators are provided with the component implementations, the component descriptions and component dependency documentation by the software developers.

The experts at iNTENCE performed manual integration test (cf. section 2.4) but also executed automatic integration tests by simulating the software functionality and its environment interactions on PC emulators (e.g., in a project at Continental Automotive GmbH).

Integration Challenges The main challenges during software integration according to the iNTENCE representatives are:

- Time pressure: After an implementation delay there is less time available for the software integration.
- Erroneous Components: Functionally defective components and/or components that cannot be compiled need to be revised by the integrator. This effort also decreases the time for integration and integration testing.
- Untested Components: Without a component test the integration test verdict is not useful. The result of the integration test can be flawed, since it is not clear if it is caused by a functional fault in one

of the components or an incompatibility between components (in case of a negative integration test verdict).

- Architecture and Requirement Deficits: Incomplete requirement documentation and low quality architecture definitions can cause very distinct component implementations by the software function developers. This can produce major incompatibilities in the architecture design phase, which become visible during the implementation phase.

- Insufficient Interface Descriptions: Missing information in the interface definitions is similar to the architectural and requirements deficits. The missing information can cause problems in the integration phase through incompatibilities or undocumented dependencies. In the worst case, a faulty interface description leads to unnecessary stub[1] development (for a wrong component dependency) or an additional missing stub (for the correct component dependency).

- Residual Stubs: In case of stubbed components, it is necessary to remove the stub when the actual component is integrated to prevent errors in the software product.

- Insufficient Collaboration and Flexibility: The integrator is playing a central role in the project team he has to bring together the results of the developers to final product. The organization of the integration phase and the flow of information during the integration phase (between integrators, developers and testers) have to be laid out in a way which enables good collaboration. Insufficient collaboration during the integration phase can also lead to decreased flexibility, since the integrator cannot react on project changes adequately.

Suggestions The results of the workshop were a set of suggestions that would improve the integration.

- Architecture Improvements: A well designed and especially a well maintained software architecture can contribute significantly to prevent errors during software integration. The implementation constraints, which are the result of the system architecture, reduce the potential of incompatible components by reducing the degree of freedom for developer for the development of a component.

- Stub Administration: The stubs, which are used to substitute components prior to their actual integration date, have to be integrated

[1] A stub denotes a skeleton implementation of a component or subsystem to simulate a test environment for integration test (cf. section 2.4).

at the time when they are required and have to be removed afterwards. This also requires the use of the correct version of the stub, since they can differ in functionality and also in the nature of their stubbed interface. To make this process more clear and less error-prone, the stubs should be organized and maintained in a specialized stub management, which also tracks the evolution of stub types.

- Improvement of Collaboration and Communication: A close connection between integrators, developers and testers is a key factor for integration. The experts at iNTENCE are suggesting an integration information and administration system to manage the delivery process for integration, i.e., the developers create tickets when a component is ready to be integrated.

1.1.2 Survey

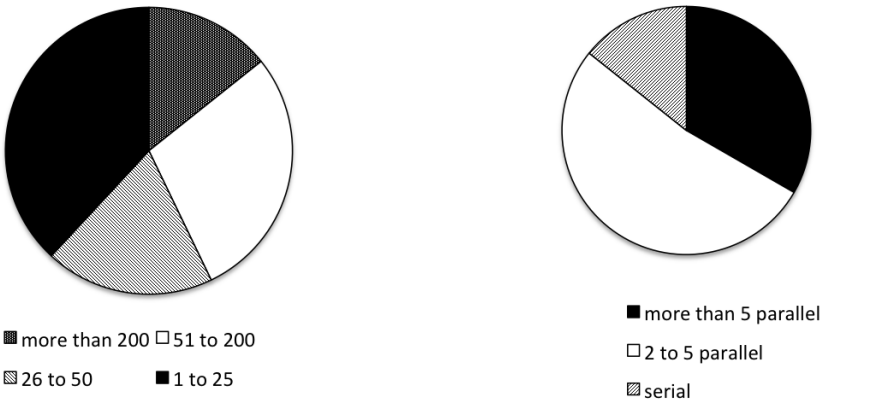
The state of the art analysis in the research project was supported through a survey. This survey was published in the industry magazine Elektronik Praxis Hafner, 2010 and was made available also through the LaS^3 website. The survey was aimed to give qualitative information about the state of the art integration process and the challenges that integrators are facing.

The survey yielded a total of 21 replies from very different industry branches and with very distinct working practices concerning integration. The project sizes in components differ strongly, cf. figure 1.3a. Figure 1.3b more than 75 percent of the participants integrate more than one component in parallel during an integration step.

According to the survey participants, the release plan for the software product is the main factor of influence for the selection of an integration strategy i.e., choosing a particular component for integration at a given point in time. The experience of the integrator came in second when it comes to determine the order integration. Satisfying component dependencies and project lead coordination play a lesser role in the selection of the strategy. Figure 1.4 shows how the participants evaluated the influence on the integration strategy.

Figure 1.5 shows the preferred integration strategies of the participants. The most favored strategies are schedule-driven and hardest-first integration. The answers are varying very strong. Every strategy has at least one survey participant marking it as being never used and at least another participant stated it as mostly used in his/her projects.

The communication between the members of an integration team was also part of the survey. It unveiled that the participating companies do not have designated workflow or communication structure to manage



(a) Project sizes in survey participants companies.

(b) Number of components per integration step.

Figure 1.3: Key figures to the participants' integration projects.

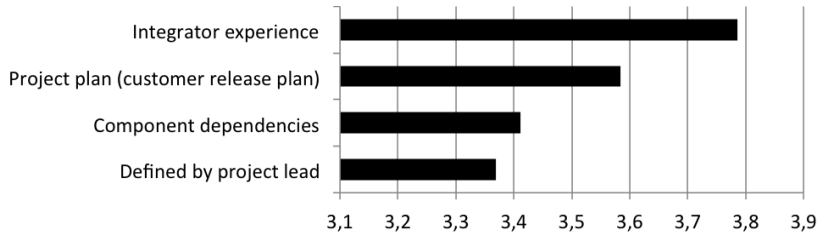


Figure 1.4: Influence on the selection of the integration strategy.

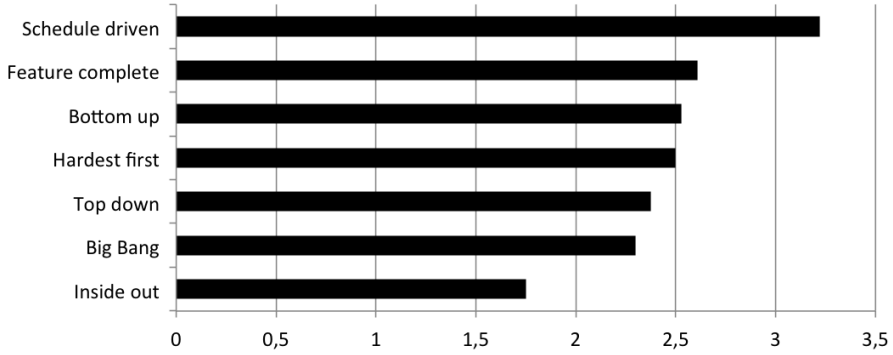


Figure 1.5: Usage of integration strategies in participant projects.

the integration process. It is common practice, to organize the integration on demand from person to person or in regular status meetings. Less common are plan-oriented management of the integration through spreadsheets or through version control software.

Figure 1.6 shows the participants experience to the most common integration problems. The top three most common errors during integration are missing or wrong documentation of the components, erroneous components and the violation of real-time constraints.

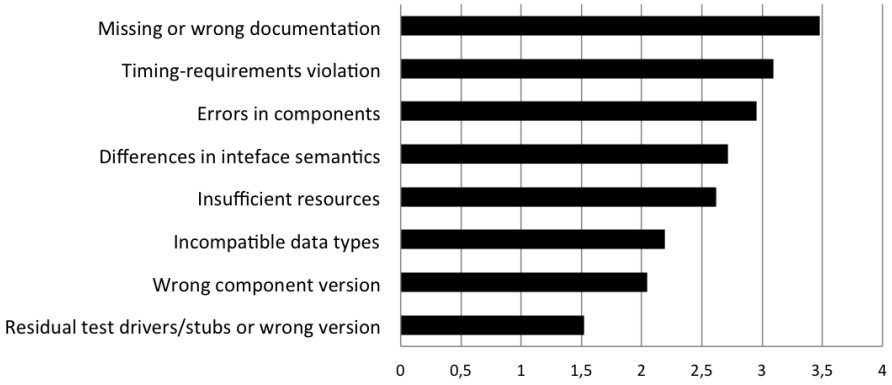


Figure 1.6: Problems during integration.

The industry experts had the possibility to make suggestions on how they would improve software integration. The following list contains selected statements of the participants:

> *'Find a balance between flexible, agile activities and a rigid integration process.'*
>
> *'Process de plan [sic], Do, Check, Act. An error is a deviation from the requirements.'*
>
> *'Collaboration with software developers: The documents* ready to integrate *and* integration test specification *are provided by the software developers. Integration is triggered by the software developers.'*
>
> *'Put everything under version control and improve communication in the team.'*
>
> *'Use tools for automatization - Create test cases during the product feature definition.'*
>
> *'Carry out the module tests in the integration test tool. This leads to an easier adaption of test artifacts.'*

'The integrated development environments are offering a lot of possibilities. Unfortunately, they are barely used due to time and cost saving measures.'

1.2 Research Problem

The workshop clearly showed that VitaS[3] addresses most of the problems that the industry is currently facing in this particular field. To address these problems, VitaS[3] proposes the implementation of the so-called virtual integration methodology. The standardized processes and task definitions of the methodology improve the software integration process.

This thesis concentrates on three main tasks that can bee seen as an implementation of the virtual integration methodology:

- Design a reference architecture and prototype for a virtual integration tool support
- Demonstrate the feasibility of interface automata as compatibility verification mechanism
- Implement an efficient integration scheduling technique

1.3 Contributions

Design of a tooling environment to support the virtual integration methodology

The virtual integration methodology is supplied with tool support during the whole development process. The thesis presents an exemplary architecture of such a tooling environment. The resulting software system represents an information system for the software integration process and its preparatory measures. The thesis also presents a prototype for the so-called integration information system on the basis of the Eclipse[2] platform.

Concept for integration schedule optimization through game-theoretic modeling

The integration scheduling process is critical for a successful integration of the final embedded software product. The thesis presents a novel integration scheduling concept which applies game theoretic concepts to achieve an optimal integration schedule under influences from the

[2] www.eclipse.org

project environment and also the complex properties of the software architecture. A prototypic implementation is embedded into the integration information system to show the practicability of the concept.

Feasibility study on interface automata for dynamic compatibility verification

Interface automata present a model to describe interaction in component-based software systems. The verification of interface compatibility is a key factor in the interface automata model. This thesis presents a feasibility study on the applicability of interface automata to verify compatibility in an early stage of development (e.g. the design phase). The goal is to reduce effort for integration testing and error correction that is caused by unidentified component incompatibilities.

1.4 Organization of the Thesis

The thesis is organized as follows. Chapter 1 includes a description of the research project VitaS^3 and the state of the art analysis (the survey and the industry workshops) which serves as motivation for this work. After this follows a description of the research questions as well as the research method.

Chapter 2 gives a general introduction on the backgrounds of software integration and its application in the automotive domain. It also gives definitions fro integration-related terms that are used throughout the thesis.

The virtual integration methodology is the key component of chapter 3. It explains its subprocesses and their application in the software development process.

Chapter 4 gives a background of information systems and shows how the integration phase can be supported by a designated integration information system.

Chapter 5 defines the architecture of a prototype of the integration information system with regard to the virtual integration process.

Chapter 6 introduces the concept of dynamic compatibility and its implications for software integration. It describes the nature of interface automata and their applicability for the verification of dynamic compatibility. The chapter presents a feasibility study on the applicability of interface automata during the software development process and closes with a case study with an industry example and an evaluation of the results.

Chapter 7 illustrates the second application of game theory in this thesis. It provides background information on integration planning and the

resulting integration planning problems are illustrated. It also features the concept of integration cost measurement and shows how integration games can be used to optimize integration schedules. The chapter closes with a case study and an evaluation of the results.

Chapter 2

Integration in the Software Engineering Context

THIS chapter gives an overview of the thesis background. It shall further give an insight on the industry practice that lead to the challenges described in section 1.2. This section also contains a set of essential working definitions that are used throughout the thesis.

2.1 Integration in the Development Process

The V-model was introduced in Boehm, 1979 and is a widely spread development methodology that extends the Waterfall-Model though validation and verification of the work products.

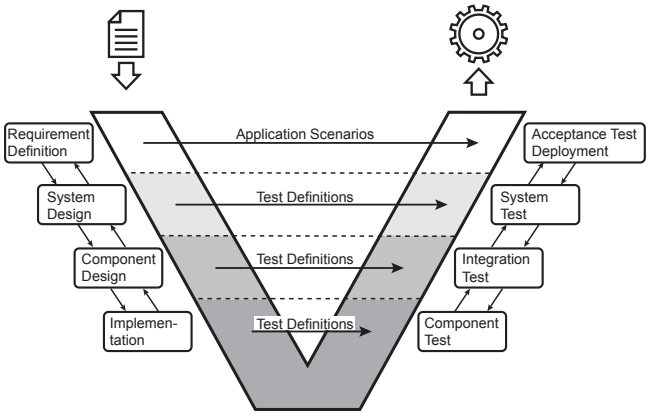


Figure 2.1: The standard V-Model. (Source: Balzert, 2008)

Figure 2.1 show the phases and their alignment over the project de-

velopment time. The left side covers the continuously more detailed development steps in which the stepwise design and the final programming of the system takes places. The right side stands for integration and test activities, in which atomic program building parts are successively built (integrated) into bigger subsystems and are tested on their proper functionality. Integration and test end with the acceptance test of the system (Spillner and Linz, 2012). In figure 2.2 the V-model is shown in its normalized form.

The development process of software systems in the automotive domain is a modification of the standard V-Model. One of the main differences to the standard V-Model (illustrated in figure 2.3) is that it incorporates a series of intermediate releases before the final release. The intermediate releases contain a predefined product extend which was negotiated with the project manager before the development of the software system started. The customer can use this early delivery to perform further test with a restricted functionally of the system.

Roles in the Integration Process

The state-of-the-art analysis at Continental Automotive GmbH and iNTENCE Automotive GmbH made it possible to define a set of role descriptions for the integration phase. The following roles are involved in the integration process. Each role has individual and sometimes contradictory incentives during the integration phase.

Customer The customer negotiates the feature content of the software product and its releases with the project manager. The customer tries to maximize the number of product features and the product quality while not giving up any flexibility during the development process (e.g. for feature additions or changes). On the other side the customer's main interest is a minimum product price and a minimum product delivery time. Opposed to the customer's requirement to be flexible in terms of product extend and time frame, he expects the product vendor not to violated the negotiated release dates.

Project Manager The project manager is responsible to deliver the product interim releases as well as the final product to the customer at fixed points in time. He has to assign the available resources for the projects. The project manager has major impact on the integration phase since he determines the available resources for the integration and the time frames during which the individual components can be integrated. The project manager has the following incentives with regards to integration.

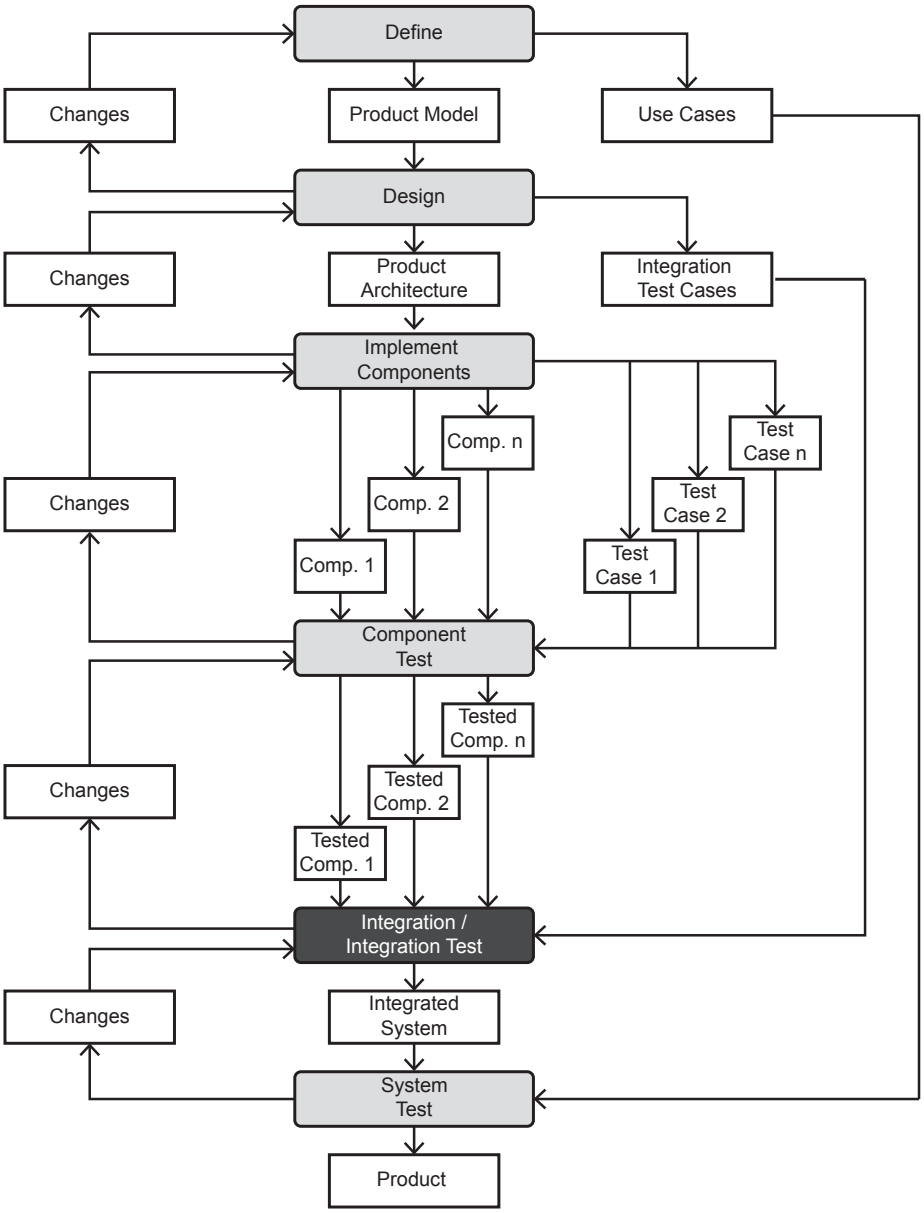


Figure 2.2: The standard V-Model in its normalized Form (The sides of the V are flattened to illustrate the sequential order of the development tasks and the iterations in case of system changes.). (Source: Balzert, 2008)

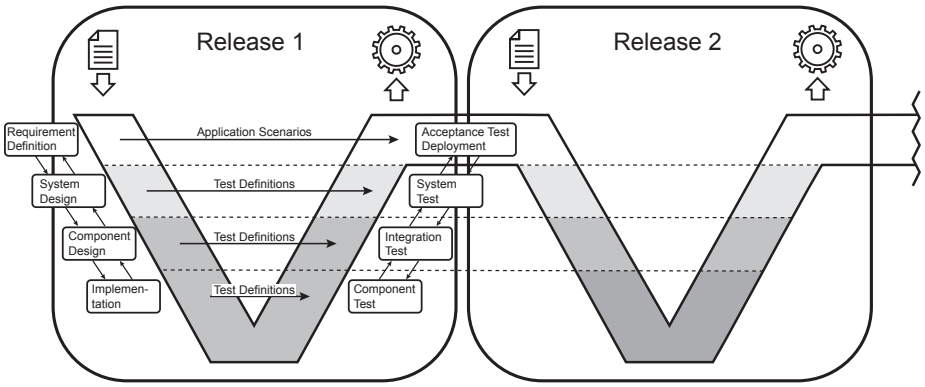


Figure 2.3: The iterated V-model.

He requires sufficient resources and development time for the development of the software system. Also, he aims to keep customer change requests during development time to a minimum. A low number of interim releases also increases the project managers flexibility during the project.

Integrator The integrator is the central roe in the integration process. He is responsible to perform the integration according to the project schedule, which is supplied thorough the project manager. The integrator received the software components after they are completely implemented and after the component test is completed successfully. The primary goal of the integrator is to perform the integration in compliance to release plan. He further aims to have sufficient resources during integration and a sufficient time buffer to resolve possible integration faults.

Integration Tester This role is often assigned together with the integrator role. The integration tester receives the previously integrated system and performs the integration test. His goals are to perform a successful integration test with a high test coverage. The test effort in the integration test rises with the number of newly integrated components.

2.2 Software Architecture

The software architecture of an automotive software system is the result of a design process in the software development process. The SEI[1] describes software architecture in SEI, 2012 as follows.

[1]Software Engineering Institute

> The software architecture of a program or computing system is a depiction of the system that aids in the understanding of how the system will behave. Software architecture serves as the blueprint for both the system and the project developing it, defining the work assignments that must be carried out by design and implementation teams. The architecture is the primary carrier of system qualities such as performance, modifiability, and security, none of which can be achieved without a unifying architectural vision. Architecture is an artifact for early analysis to make sure that a design approach will yield an acceptable system.

In the standard V-Model, the system architecture is created by deriving the system requirements into dedicated subsystems. These subsystems, can be the input for subsequent design steps that refine the system architecture further. For example, the EAST-ADL2 automotive architecture description language defines two types of system architecture (the functional analysis architecture and the functional design architecture), which are created in different phases of the development process.

2.2.1 Component-based System Architecture

A component-based system architecture is a description of a software system that is build from components. These components are connected to each other in order to provide the system's features. According to Crnkovic, 2005, the four key factors for implementing a component-based system architecture in embedded system software development are:

1. Reusability: Components may be reused in different systems
2. Substitutability: Different implementations of a component may be used
3. Extensibility: The functionality of individual components may be increased
4. Composability: Components may be composed to provided a desired functionality

The definition of a component is formulated by Brown in Brwan, 1997:

> A component is an independently deliverable piece of functionality providing access to the services through interfaces.

In contrast to Brown's rather general definition Kopetz gives a technical definition of a component in Kopetz, 1998. He states that an ideal component is a

> ... self-contained computer with its own hardware (processor, memory, communication interface, interface to the controlled object) and software (application programs, operating system), which performs a set of well-defined functions within the distributed computer system.

When applied to the automotive domain, Kopetz's definition of a component matches an integrated control system, for example the engine management system. In contrast to Kopetz's definition of the definition of Szyperski is focused on software components can be found in Szyperski, Gruntz, and Murer, 2002:

> A software component is a unit of composition with contractually specified interfaces and explicit context dependency only. A software component can be deployed independently and is subject to composition by third parties.

2.2.2 Components Dependencies

Following Szyperski's definition of a component, the dependencies between components are a central concept of component-based software architectures. The satisfaction of these dependencies must then be one of the main goals in order to build a functioning system. Ma, Wang, and Lu, 2006 give a brief overview of different dependency types:

Data dependency Data dependency, which is produced by data integration between different COTS[2] components. In general, data dependency represents that data is defined in one component, but used in another one.

Control dependency Control dependency, which is produced by control integration in component-based systems, is not an explicit dependency. Control dependency is caused by broadcasting, remote procedure calls or by general passing.

Time dependency Time dependency represents that the behavior of one component precedes or follows the behavior of another one in component-based systems.

State dependency State dependency represents that the behavior of a basic component will not happen unless the system, or some part of the system, is in a specified state.

[2] A component off-the-shelf or commercial off-the-shelf is a standardized component that has has a very limited customizability.

Cause and effect dependency Cause and effect dependency represents that the behavior of one component implies the behavior of another component.

Input/Output dependency Input/Output dependency represents that a component requires/provides information from/to another component.

Context dependency Context dependency represents that a component requires a designated execution environment.

Interface dependency Interface dependency is produced by user interface integration. Usually, the interface-event dependency is the main dependency form in component-based systems. When one component needs another component to do something, it first sends a message to trigger an event through its interface, then the event activates another component.

2.3 Compatibility

Compatibility is defined in the ISO 8402 standard as ”Eignung einer Einheit unter spezifischen Bedingungen zusammen benutzt zu werden, um relevante Forderungen zu erfuellen.”. The MobilSoft research project defined compatibility by dividing it into three major aspects:

Consistency Consistency describes the logical and functional correctness of the component. This functionality can be achieved in cooperation with other tasks.

Interoperability Interoperability is defined as the faultless cooperation between system components, which includes functional as well as dynamic aspects. Examples are interface semantics and communication with other components.

Conformity Conformity describes how the components meet requirements, which are deducted from the system's architecture specification (e.g. operating system, protocol or interface syntax).

Interoperability and *conformity* are subject of the integration test. *Consistency* is tested in the component test.

2.3.1 Static Compatibility

Static or structural compatibility is focused on the compatibility of interface syntax of interconnected components as well as architectural and

environmental constraints. When ports of components are used to transmit signals between components, their characteristics are required to match. Examples of characteristics are filter type, signal quality or physical unit. A component's physical properties describe how well it to conform to requirements of physical nature. These properties are strongly linked to the hardware platform.

Communication between software components in embedded systems is realized as exchange of signals (sender-receiver concept). These signals can have several properties apart from their 'content':

Filter Information if the signal is filtered in any way can affect the way it will be used in the receiving component.

Quality Quality information can influence to which extend the receiving component will rely on the signal.

Update Frequency If the receiving component is dependent on up-to-date values the sender needs to have a sufficient update frequency.

Analog/Digital The conversion into a digital signal can change the way the signal needs to be treated by the receiver.

Conversion It may be necessary to carry out a conversion to receive the correctly scaled value from a fixed-point parameter.

2.3.2 Behavioral Compatibility

Behavioral compatibility between components is achieved when the behavior of the involved components provides correct interaction. This means, that signals (or information in general), which is required by a specific component at a specific point in time (or at a specific state) in the execution of the software function, are provided by another component.

In real-time systems the timing is the most important property of the system. Therefore, the timing properties of a system's components play a key role. Below, some examples for properties, which influence a components temporal behavior, are listed:

Task Allocation Tasks in real time systems are structures that gather atomic software functions in order to be executed in a designated time slot.

Activation The components activation denotes the event that triggers the execution of the components functionality.

Recurrence Components inherit the recurrence of their execution from the task properties. This can be sporadic, e.g. for event-triggered tasks, or recurring in a constant time for time triggered tasks.

Execution Time The execution time of the functionality of a component plays a key role to establish a system behavior that satisfies the real-time requirements of embedded control systems in the automotive industry.

Deadline The deadline of a task denotes the designated point in time when the execution of the task shall be completed relative to its activation. Violations of task deadlines influence the schedulability of the task systems an can lead to the violation of the systems real-time requirements.

Priority Tasks in an embedded system have priorities assigned to them. The priority of a task is the selection criteria to choose the next fast for execution for certain scheduling algorithms.

Behavioral compatibility of components in embedded systems can not be reduced on the temporal aspects but they are by far the most complex part. For an in-depth description on temporal properties of embedded systems and task scheduling see Schäuffele and Zurawka, 2010.

2.4 Integration Testing

The testing of the interconnections and the interaction between software components is called *integration testing*. According to Eickelmann in Eickelmann and Richardson, 1996, it insures the consistency of component interfaces and whether the components pass data and control correctly, which results in successful integration of dependent components.

Integration testing ensures the correct interaction between components. It is the test of the static and dynamic compatibility, whose aspects where described in detail in the previous section. Software integration and integration testing are often used synonymously in literature. The difference between these two terms is distinct. Integration denotes the practice of combining a specific set of components into a system. The goal of integration is to carry out this combination process in the most effective way. Integration testing requires a prior integration step as test subject. Its goal is to uncover incompatibilities between the set of component, which were combined in the software integration process.

Figure 2.4 shows an example integration test setup. The *system environment* represents the current set of components in the system. The integration test of these components its already complete. The integration test is focused on the interconnections of the *system environment* and the *component under test* (CUT) . *Test drivers* are used to inject test stimuli in the setup and to retrieve the test results. Any components that

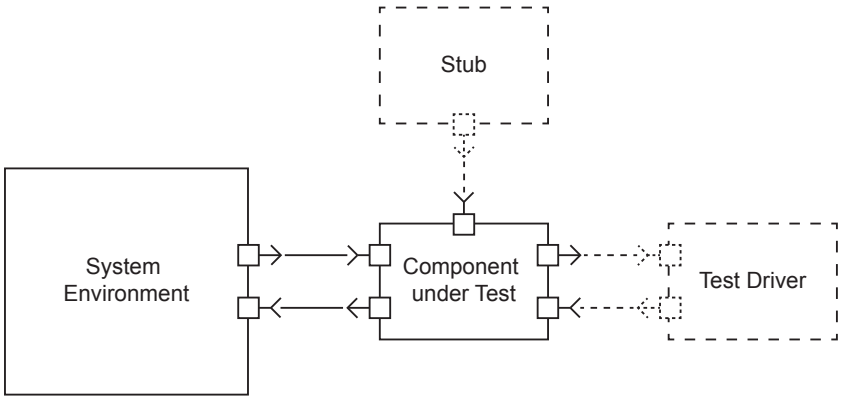


Figure 2.4: Integration test setup. The compatibility between the component and its target system environment is tested. The stub simulates unsatisfied dependencies of the component. The test driver enables the tester to inject test stimuli and inspect the test results.

are required by the CUT but are not present in the system environment are substituted through *stubs*.

2.5 Integration Strategies

The integration of software components can be distinguished by different integration strategies which are used in various forms and have different strength and weaknesses. The used integration strategy has a great impact on the software integration process and varies from company to company. In the following section the different strategies are presented and compared to each other. The evaluation is focused on the application of the integration strategies in the automotive domain with its particular domain characteristics. In Gao, Tsao, and Wu, 2003 and Binder, 1999 well-known integration strategies are presented. These methodologies use the functional decomposition, which is often expressed in a tree structure. Given the functional decomposition tree, four different approaches can be used for integration.

Evaluation

The following evaluation criteria are used to characterize the integration strategies. They were defined together with industry experts in the course of VitaS[3] and reflect the common requirements towards integration plans in development projects.

Testability Testability defines the effort which is necessary to carry out the integration test for the given integration strategy. It is therefore also a measurement of the test coverage and ultimately the product quality since test resources are limited in the industry.

Stub Avoidance Stub avoidance represents the ability to avoid the development of stubs during the integration and integration test. Use of stubs requires effort for development as well as in the module test. A low number of stubs reduces the cost for integration testing.

Flexibility Flexibility defines how well the integration strategy can be adapted to changing environmental influences. A high grade of flexibility is favorable since changes in the system architecture or release dates or reengineering due to software errors need to be reflected in a change of the integration plan.

Mapping on Release Plans Since the release plans are often laid out to deliver a set of product features at a given point in time during the development process, their design/structure needs to be represented in the integration plan as well. Mapping on release plans represents how well the integration strategy can be mapped onto project release plans.

Integration Timeframe The integration timeframe defines the overall time, which is necessary for integration when the given integration strategy is used. The shorter the overall integration timeframe, the higher the evaluation score for this criteria.

Granularity Granularity represents how the integration is carried out with respect to the connectivity of the components. It defines the number of not directly connected subsystems during integration. This has an influence on the ability to deliver a complete feature before its planned release date. A low level of granularity will therefore be rated as positive in the evaluation score.

Degree of Formalization The degree of formalization defines to which extend the execution of the integration sequence can be formalized. This influences the ability to automate the integration and to validate the integration automatically.

All integration techniques are evaluated according to the criteria from above and the results are displayed in a short overview for each technique through radar charts. The evaluation score ranges from 0 (no satisfaction of the criterion) to 5 (full satisfaction of the criterion) points.

2.5.1 Big Bang

In Beizer, 1984, big bang integration is characterized by Beizer as follows:

> *In its purest (and vilest) form, big bang testing is no method at all - 'Let's fire it up and see if it works!' It doesn't of course.*

Big bang integration is an integration strategy, that consists of one single integration step. All components are built and brought together in the system without regard for inter-component dependencies or risk. This leads to difficulties in fault identification. If a failure is encountered, all components are under suspicion.

Nevertheless there are scenarios in which big bang integration is reasonable:

- The system architecture is stable, i.e. only few components were added or changed since the last passed test. This makes the integration test with big bang integration practicable.
- The system is small, i.e. the low number of components makes an integration test possible.
- The system is monolithic and can not be exercised separately.

If none of these situations is present, big bang integration is not a proper solution and it usually creates more problems than it solves.

Evaluation:

1. **Testability** Big bang integration has its major weakness in the testability of the integrated system. All components have to be built before integration testing and faults can not be detected until a very late stage of the development process. It is very difficult to debug in case of an error, since the error location is not known precisely. Every component is equally suspect of causing the error. Since the integrated system can only be stimulated from outside, it is possible that not all integration faults can be detected.
2. **Stub Avoidance** Big bang integration does not require any drivers or stubs since all of the components are available at the same time.
3. **Flexibility** Big bang integration can be considered flexible in terms of being able to add or delete components because the integration itself is not carried out in a sequential (incremental) way. On the other hand, it is not possible to deliver an intermediate release to the customer.

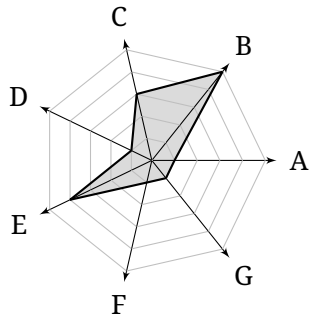
4. **Mapping on Release Plans** Big bang integration can only be used if the release plan does not include intermediate product releases. Since intermediate releases are a very common practice in the automotive software development, big bang integration is not satisfying this need.

5. **Integration Timeframe** There is a chance that the integration is carried out very quickly if there are no integration faults. However, when integration faults are discovered, the integration phase is expanded dramatically due to the limited testability.

6. **Granularity** It is not possible to deliver or to demonstrate features or subsystems of the product before its final assembly.

7. **Degree of Formalization** As mentioned by Beizer, 1984, big bang integration does not specify any methods or supplementary processes at all.

Table 2.1: Evaluation Result: Big Bang Integration

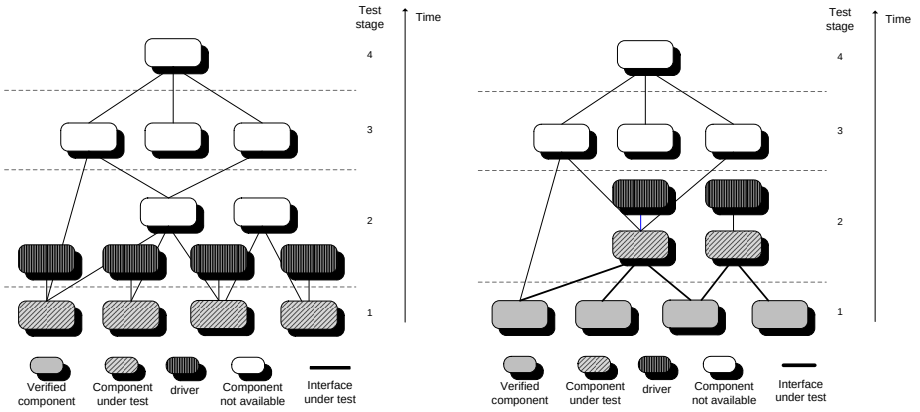


	Criteria	Score
A	Testability	1
B	Stub Avoidance	5
C	Flexibility	3
D	Mapping on Release Plans	1
E	Integration Timeframe	4
F	Granularity	1
G	Formalization	1

2.5.2 Bottom Up

Bottom up integration achieves stepwise verification of the interfaces between tightly coupled components. Components with the least number of dependencies are integrated first. When these components pass, their drivers are replaced with their clients and another round of integration begins. Figure 2.5 shows a schematic illustration of the bottom up integration procedure. This strategy is suited for responsibility-based designs and systems of components with stable and robust interface defi-

nitions. Bottom up integration is also widely used if a project is started from scratch.



(a) First stage. (b) Second stage.

Figure 2.5: Example of bottom-up integration in two stages.

Evaluation:

1. **Testability** Testing of components can only be done on the same stage, there is no explicit testing of interfaces over several stages. Furthermore, it can be difficult to stimulate lower components at a late stage of integration.

2. **Stub Avoidance** A significant amount of driver implementation and test effort is needed to perform bottom up integration.

3. **Flexibility** The order of component integration can only be changed within a certain layer.

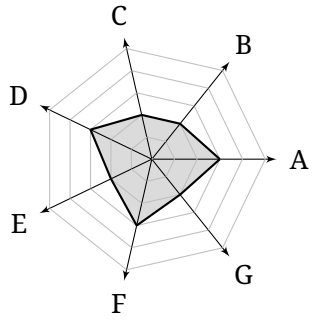
4. **Mapping on Release Plans** Bottom up integration can only be mapped on a release plan, which consists not of individual features of the product but rather of technical separated parts. For example, all hardware drivers could be delivered in a first release.

5. **Integration Timeframe** It is possible to execute integration on a single layer in parallel to reduce the integration timeframe.

6. **Granularity** Demonstration or release of system functionalities or complete subsystems is not possible until the final stage of the system is integrated because it incorporates most of the essential control functions.

7. **Degree of Formalization** The formalization of the method is extended in comparison with the big bang approach. It requires an explicit classification of the components in order to separate them into the different layers.

Table 2.2: Evaluation Result: Bottom Up Integration



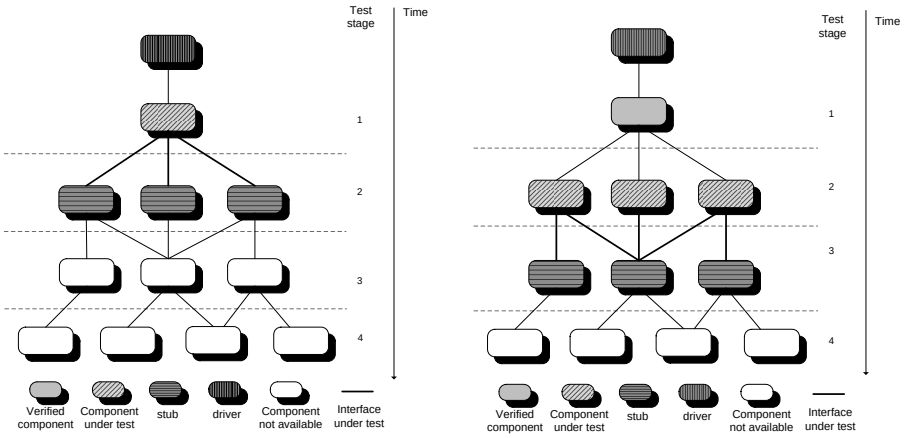
	Criteria	Score
A	Testability	3
B	Stub Avoidance	2
C	Flexibility	2
D	Mapping on Release Plans	3
E	Integration Timeframe	2
F	Granularity	3
G	Formalization	2

2.5.3 Top Down

Dependencies between components are important when using top down integration but the order of integration is reversed in comparison with the bottom up pattern. The top level component is coded first and the unavailable lower level components are implemented by stubs. After that, the stubs are replaced stagewise by full implementations and the next lower level of components is stubbed. Figure 2.6 shows a schematic illustration of the top down integration procedure.

Evaluation:

1. **Testability** Like bottom up testing, the integration test can only be done on the same level. There is no explicit testing of interfaces over several stages. Furthermore, it can be difficult to stimulate lower components at a late stage of integration. Interface problems between hardware, system software and the exercised software are detected in a late stage of the integration process because low level components are tested in the last stage. Interface problems are often expected here. A simple test of error handling in case of faulty return values is possible because they can be easily created by stubs.



(a) First stage.

(b) Second stage.

Figure 2.6: Example of top-down integration in two stages.

2. **Stub Avoidance** A significant amount of stub implementation and test effort is needed to perform top down integration.

3. **Flexibility** The integration order of the component can only be changed within a certain layer.

4. **Mapping on Release Plans** Top down integration can only be mapped on a release plan, which consists not of individual feature of the product but rather on technical separated parts. For example, the user interface could be delivered in a first release.

5. **Integration Timeframe** A parallel development of components is possible, components of one stage are coded and tested simultaneously.

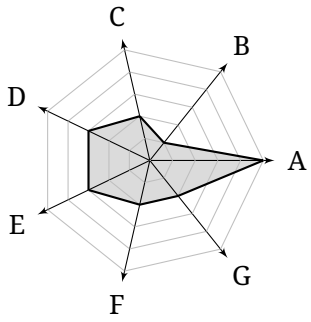
6. **Granularity** High level components typically implement essential control functions. Top down integration exercises these components first, so a demonstration of the system is possible at an early stage of integration.

7. **Degree of Formalization** Like bottom up integration, the formalization of the method is extended in comparison with the big bang approach. It requires an explicit classification of the components in order to separate them into the different layers.

2.5.4 Outside In

Outside in integration testing is a combination of bottom-up and top-down integration pattern. The integration is started both from the hard-

Table 2.3: Evaluation Result: Top Down Integration

		Criteria	Score
	A	Testability	5
	B	Stub Avoidance	1
	C	Flexibility	2
	D	Mapping on Release Plans	3
	E	Integration Timeframe	3
	F	Granularity	2
	G	Formalization	2

ware and from the user/environment interaction side simultaneously. Figure 2.7 shows a schematic illustration of the outside in integration procedure.

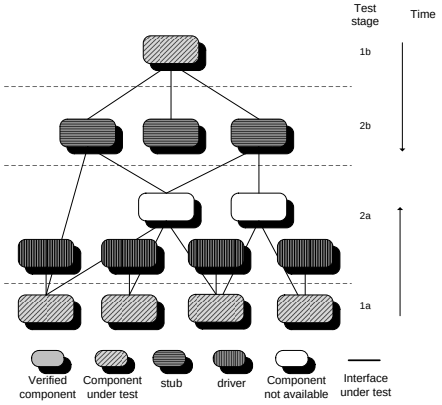


Figure 2.7: Outside in integration in the first stage

Evaluation:

1. **Testability** The outside in integration enables simple test of error handling in case of faulty return values because they can be created by stubs (in top down case) and simple test of error handling in case of faulty input values because they can be injected by drivers (in bottom up case).

2. **Stub Avoidance** Because the integration process starts from two fronts the number of required stubs is increasing.

3. **Flexibility** The integration order of the components can only be changed within a certain layer.

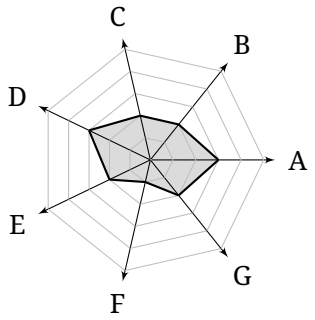
4. **Mapping on Release Plans** Inside out integration can only be mapped on a release plan, which consists not of individual feature of the product but rather of a technical separated parts. For example, the user interface or all hardware drivers could be delivered in a first release.

5. **Integration Timeframe** Because inside out integration is able to be executed in parallel, the integration time can be reduced significantly.

6. **Granularity** Essential control functions are exercised in an early state of integration so a demonstration of the system is possible in a low stage of integration.

7. **Degree of Formalization** Inside out requires an explicit classification of the components in order to separate them into the different layers. But it does not define the integration order within a certain layer.

Table 2.4: Evaluation Result: Outside In Integration



	Criteria	Score
A	Testability	3
B	Stub Avoidance	2
C	Flexibility	2
D	Mapping on Release Plans	3
E	Integration Timeframe	2
F	Granularity	1
G	Formalization	2

Chapter 3

Virtual Integration

CHAPTER 2 and especially the presentation of current integration techniques in section 2.5 showed that currently there is no approach which fully addresses the challenges that arise from the integration survey or the expert workshop (cf. sections 1.1.2 and 1.1.1). These challenges eventually lead to the funding of the research project VitaS^3. This chapter introduces the virtual integration methodology which is the central work product of VitaS^3.

3.1 Introduction

Previous work on the topic can be found in Giusto et al., 2002 and Kanajan, Zeng, Pinello, and Sangiovanni-Vincentelli, 2006. They interpret the term virtual integration as a methodology to provide the software developer with means to perform an integration of functional behavior of components. Their approach aims to provide an early functional verification of the software system, a functionally oriented impact of change analysis and also design space exploration [1]. The functional aspect of their definitions is also tightly coupled with the hardware platform of the software system, i.e. the virtual integration is carried out between functional models of software components as well as hardware models.

In the course of the research project MobilSoft the term virtual integration was described from a different point of view. The authors define virtual integration as the process of combining interface definitions of single development elements into a system. They state further that the compatibility of individual components between each other is examined formally in the course of the virtual integration. Also, the virtual inte-

[1] Design space exploration denotes a development technique that assist software developers (especially in embedded systems) in choosing between design alternatives. For further reading see Kuenzli (2006).

gration can take place on all modeling levels. Both approaches share the idea of preparatory measures for software integration in the form of the so-called virtual integration. The single artifacts of the software system (or components of the software system) are described by structural, functional and behavioral models. The software integration is then executed virtually through composition of these models.

The two approaches above present definitions of virtual integration, which are centered on the compatibility and the guidance on the composition process of a software system. The results of the expert interviews and the survey (cf. section 1.1.1), which were conducted in the research project VitaS^3, show that virtual integration needs to address the organizational nature of a software development project. This extension of its scope increases the impact of the methodology on the software development process. It takes into account that software development is carried out in an environment of restricted resources in an organizational environment, e.g. distributed project teams. The two key resources for software development projects are manpower and time. There are some auxiliary resources like restricted access to certain development tools or specific hardware, but these are negligible in the general case. VitaS^3 showed that these restrictions have to be taken into account for software integration and not only during the implementation.

The approach of Kanajan or Giusto (cf. Giusto et al., 2002; Kanajan et al., 2006) is a general approach on how to design the hardware as well as the software embedded systems with the goal of a successful integration in mind. This introduces fundamental changes in the software development process. The aim of VitaS^3 was to enhance the established industry practice to improve the software integration process.

The work, which was done in the MobilSoft project offered the basis for this approach. The methods for compatibility checking, where extended through methods for integration plan optimization, integration administration, integration monitoring and automated builds.

These parts are concluded in a comprehensive methodology, the *virtual integration methodology*. Each part of the methodology was implemented in a prototype to form a tool chain that supports virtual integration throughout all its phases. In the following, the virtual integration methodology is described in detail.

3.2 Methodology

Virtual Integration means to perform integration activities virtually before they are finally carried out. *Integration* in software engineering means the process of assembling a whole software system from compo-

nents (cf. section 4.5). The term *Virtual* means in this context, that the inspected system is not completely implemented. Exemplary reasons for this could be: An early development stage with only specifications of the system available, a new subcomponent is added to an existing system or the change of the systems environment. In each of these scenarios the elements, which are new or subject to change can have an influence on the integration of the resulting software system.

The main goal of this methodology is to describe when integration-relevant information is produced during the software development process. Furthermore, it defines the methods which leverage this information to improve software integration. Each of these methods is further defined through input and output work products, assignment of designated roles and tool requirements.

The virtual integration methodology is structured into five parts:

1. Compatibility verification
2. Integration planning
3. Integration monitoring
4. Integration administration
5. Build automation

Figure 3.1 shows the location of the virtual integration methodology parts aligned to the iterated V-model. Compatibility verification (1) is performed on the left side of the *V* in every iteration and on each development layer [2]. The results are used to refine the system design in order to achieve compatibility between the components. Integration planning (2) is carried out at the same recurrence as compatibility verification. Integration planning results, i.e. integration plans, are used during the corresponding integration steps on the right side of the V-model. Integration monitoring (3) is a continuous process that starts when the first integration steps are performed and ends after the system integration of the final product. Integration administration (4) is performed during the integration phase of each V-model iteration. The build automation process (5) of the virtual integration methodology is performed after the implementation phase in a V-model iteration is completed.

This section defines the essential parts of the methodology, their purpose as well as their arrangement with respect to the V-model development process.

[2] Each of these layers represents a level of abstraction of the system architecture.

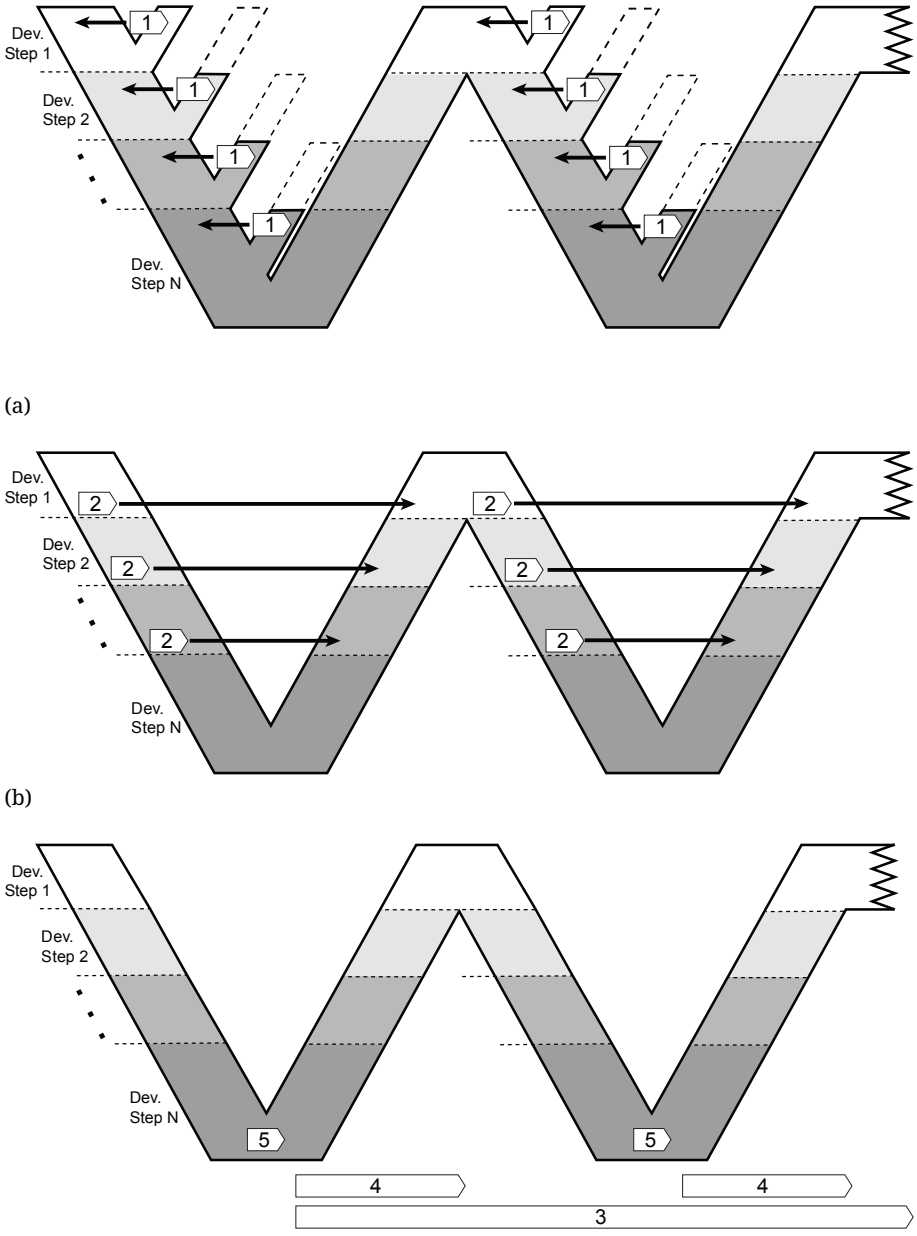


Figure 3.1: Alignment of the five virtual integration disciplines to the iterated V-model. 1) Compatibility verification 2) Integration planning 3) Integration monitoring 4) Integration administration 5) Build automation

3.2.1 Compatibility Verification

The compatibility verification subprocess is executed after every system design iteration on the left side of the V-model. A description of the system architecture is the result of each system design step on each level of abstraction. This system architecture is examined according to compatibility criteria (cf. section 2.3). These criteria correspond to the test criteria for the integration test on the right side of the V-model. Figure 3.2 shows the details of the compatibility verification process. The task structure of the existing integration and integration test activities is reproduced virtually within the system design layers. Since the integrator is responsible for the integration of the actual software product, he also performs the compatibility checking task.

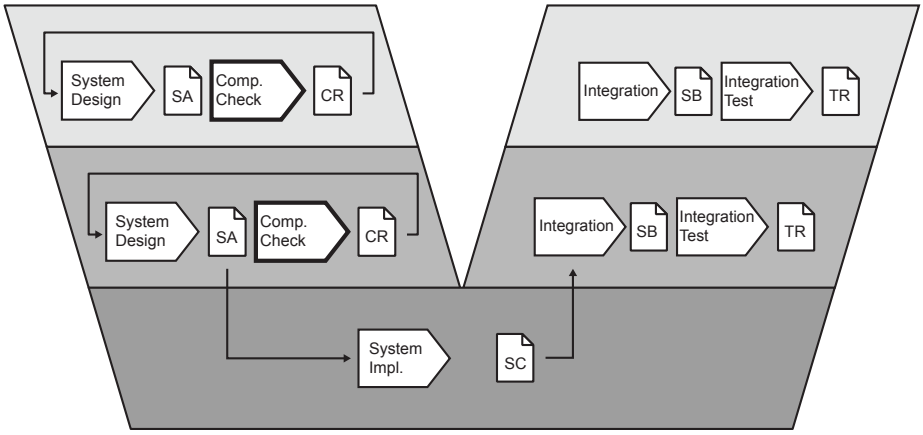


Figure 3.2: Compatibility Check process. SA = System Architecture, CR = Compatibility Check Result, SC = Source Code, SB = Software Build, TR = Test Report

The integrator uses adequate compatibility checking tools to identify flaws in the system architecture. The result of this checking process is a compatibility check report. The compatibility check report contains a detailed description of incompatibilities in the system architecture. See section 2.3 for further information on the factors of influence on component compatibility. If incompatibilities are discovered, the compatibility check result provides input for a rework of the system architecture.

3.2.2 Integration Planning

Like the compatibility check, integration planning is performed for each level of abstraction in left side of the V-model. Each of the integration planning instances corresponds to an integration instance on the right

half of the V-model. Figure 3.3 shows a development over three abstraction levels, two design levels and an implementation level. Each of the iterations in the iterated V-model represents the development of a single release of the software product. The start of a single V requires a release plan (cf. RP in figure 3.3), which includes the required features of the release as well as the due date of the release. This release plan is required to produce the project plan (cf. PP in figure 3.3).

The release planning task is mainly a negotiation between the project manager and the customer. The nature of the product has been agreed upon in the project contract. They determine which part of the system shall be delivered at which point in time. The result of this task is a release plan, which allocates the requested functionalities to a specific point in time during the project's development time frame.

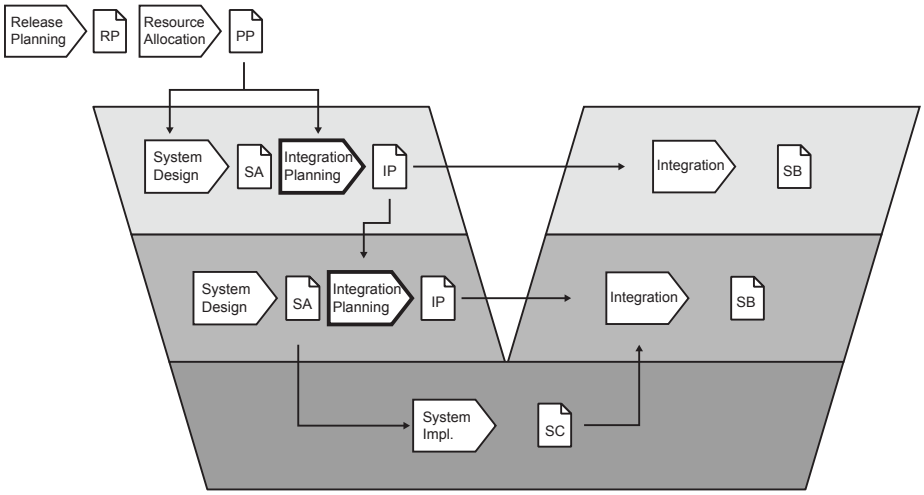


Figure 3.3: Integration Planning process. RP = Release Plan, PP = Project Plan, SA = System Architecture, IP = Integration Plan, SC = Source Code, SB = Software Build

The project plan defines which resources are available during development time. The project plan is the input for the first development layer of each V-model iteration. The resulting system architecture is the input work product for the integration planning task. The project plan is also necessary as a second input work product. It defines which resources are available later during the integration phase just as it defines which resources are available during system design or implementation. The result of the integration planning task is an integration plan (IP) or schedule. It defines which component of the software product shall be integrated at a specific point in time during the integration phase.

The integrator can develop the integration schedule manually based

on his individual experience. Alternatively he can also employ tool-based automatic integration planning functionalities to generate parts of the integration plan or completely generated integration schedules. After the schedule is set, the integrator can determine which and when components need to be stubbed. This task is also supported by the expertise from previous integrations. If the integrator discovers that the integration schedule is not feasible he may propose changes in the release plan, the project plan to the project manager.

The integration plan is used later on the right side of the V-model to carry out integration activities on the corresponding layer of abstraction.

3.2.3 Integration Monitoring

The integration monitoring is a supporting task, which is executed parallel to the integration planning and the integration.

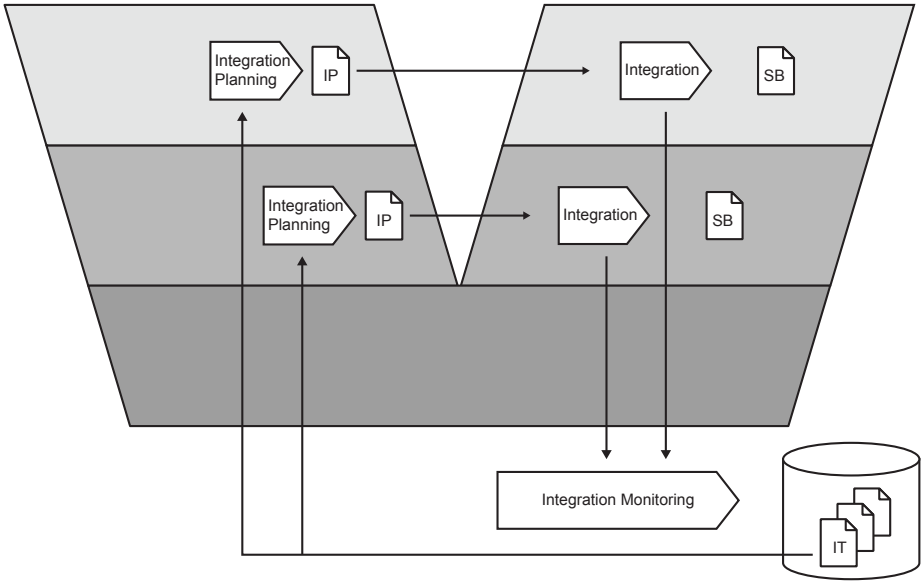


Figure 3.4: Integration monitoring process. IP = Integration Plan, SB = Software Build, IT = Integration Trace

Creation and storage of lessons learned is often already performed in industry projects (cf. Schorer, 2010a). These documents are mostly non-formal and do not include integration specific information. Integration monitoring in the virtual integration methodology is to extend the lessons learned to cover the integration phase as well. Also, the lessons learned need to be stored in a common format that can be used during integration planning of future software development. The integrator stores

information about already integrated systems, the according integration schedule and additional information in a database. This database is accessible during future integration planning phases.

Additionally, it serves as a general monitoring platform for the integration activities. The project manager can use it to get a convenient overview of the integration performance and can adjust the project plan accordingly. The goal of integration monitoring is to make software integration measurable and therefore comparable. This is done through the definition of a measurement method and an according measurement process. An example of an implementation of such an integration measurement technique is shown later in section 7.2.

3.2.4 Integration Administration

The state of the art research from **sotaconti**; Hafner, 2010; Mottok, 2009; Schorer, 2010b showed clearly that the administration of software integration is currently depending heavily on the individual integrators. In integration scenarios with complex system architectures or with more than one integrator, a self-service management and administration is no longer sufficient.

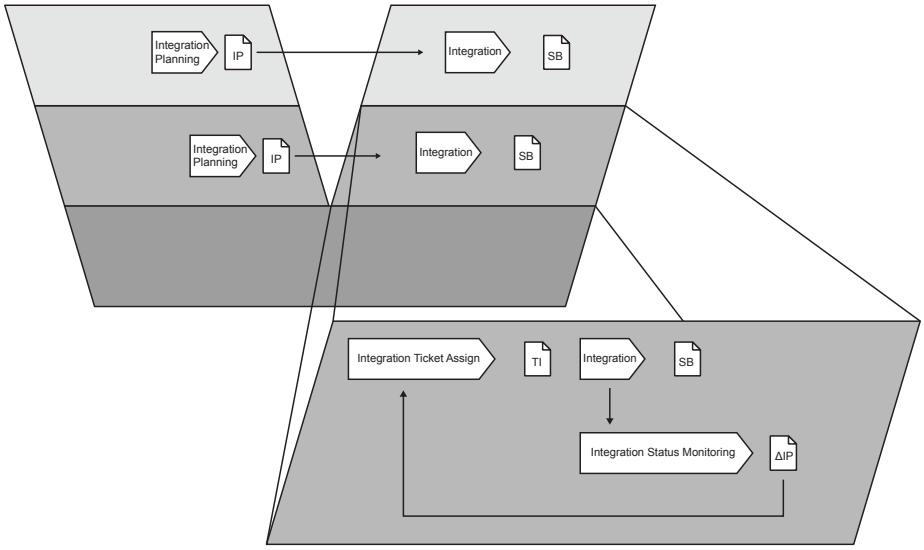


Figure 3.5: Integration administration process. IP = Integration Plan, SB = Software Build, TI = Ticket for Integration, ΔIP = Changed Integration Plan

Integration administration supports the execution of an integration plan. It represents a mapping between the integration plan and an ex-

plicit integration task or integration ticket to the integrator. This assignment can be carried out manually through the project manager or the system integration lead. Since the virtual integration proposes the fully automated generation of integration plans, it is also possible to automate this assignment step.

The second purpose of integration administration, apart from assigning and triggering actual integration work, is to continually gather status information during the execution of the integration plan. In contrast to integration monitoring which stores the data for future purposes (e.g. reuse of integration sequence parts), the information that is gathered in integration monitoring serves only operative purposes. This is necessary to reconfigure the integration plan case of delays during the integration or negative integration test results. Changes in the project plan are another common case that lead to a reconfiguration of the integration plan.

3.2.5 Build Automation

The last step in the virtual integration methodology towards a flexible, risk-aware and high quality software integration is the introduction of a build automation process.

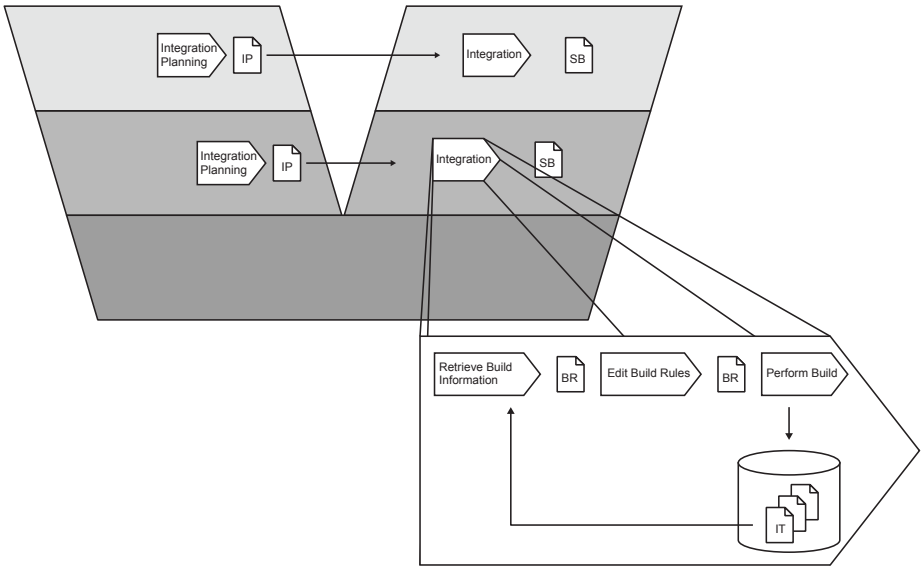


Figure 3.6: Build automation process. IP = Integration Plan, SB = Software Build, BR = Build Rules, IT = Integration Trace

Software systems in the automotive world are characterized though a high grade of software reuse. This reuse leads to a high rate of repetitive integration activities in the development of different software prod-

ucts. The integration monitoring process stores information about past software integration phases. Whenever an identical set of components needs to be integrated, it is possible to use this information to automate the integration.

The prerequisite to perform build automation is a complete identification of the components dependencies. The dependency analysis methods that are used in the integration planning process to generate feasible integration schedules can be reused in the bold automation step. The dependency information is completed with a detailed build description of each component. The build description defines how the component is build, i.e. which tool needs to be used or what build parameters are necessary for the desired target platform.

Chapter 4

Integration Information System

THE virtual integration methodology, as described in chapter 3, is only applicable in an industry setting if it is supported by an efficient tooling environment. Figure 4.1 shows the main areas of the research project VitaS[3]. An appropriate tooling is necessary for each aspect of the virtual integration (see 7 in figure 4.1).

The main focus of this tool chain is to provide the integrator with information to perform a successful integration of the software system. This information is used on the left side of the V-model[1], when preparatory actions for the integration are made, and also to administer, monitor and support the integrator during the course of integration. In this section the current state of creation, use and administration of integration-relevant (see definition 2) information in an industry setting is explained. Afterwards, the shortcomings of the current state are described in consideration of the needs of the application of the virtual integration methodology. The section closes with a description of the Integration Information System, an information system that addresses storage, administration of integration-relevant information as well as user support mechanisms during the integration phase. After an introduction of the theoretical background of information systems, the requirements for the implementation of this information system are defined and its design is described conceptually.

[1] See chapter 4.5 for a description of the iterated V-model, which is a specialized implementation of the V-model.

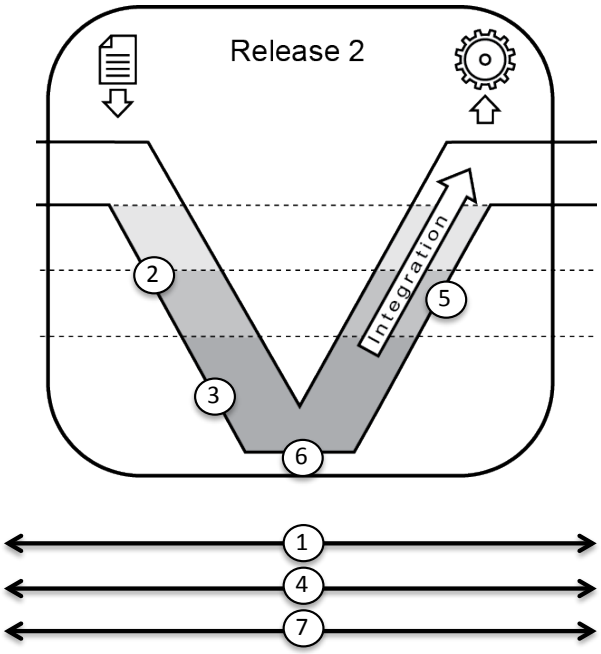


Figure 4.1: The research project VitaS[3] and its main research areas, aligned to an instance of the iterated V-model (see section 2). 1: Top level integration support methodology; 2: Compatibility verification on architecture layers; 3: Optimization of integration plans; 4: Integration monitoring; 5: Integration Administration; 6: Automated Integration; 7: Tooling

4.1 Definitions

This chapter will use a set of working definitions, which are described in the following section.

4.1.1 Information

The term information is not universally defined in literature and different definition approaches towards this term exist. Rowley approaches a definition from a knowledge-management point of view, whereas Zehnder has a more database oriented background for his definition. Schucan and especially Kuhlen define information from an information scientific point of view.

According to Rowley, 2008, information is contained in descriptions, and is differentiated from data in that it is "useful". "Information is inferred from data", in the process of answering interrogative questions (e.g., "who", "what", "where", "how many", "when").

Zehnder defines information as a usable answer to a specific problem (translated from Zehnder, 2005).

Christian Schucan defines information as the process of change that leads to an increase in knowledge and a useful change of available abstract structures (1) through additional data and/or abstract structures or (2) through additional structures or (3) through additional use of already available abstract structures. Information can trigger rational actions and/or change the interpretation of knowledge (translated from Schucan, 1999). Kuhlen summarizes the different definition approaches in Kuhlen, Seeger, and Strauch, 2004. He states that the term information does not originate from data, but from knowledge. Information does not exist as object itself. Information is a reference function. Information can only be received through a represented/coded form of knowledge. Information does not only reference knowledge representations but unfolds this meaning only with respect to its current usage context. [...] One can only talk about information in its current usage context with respect to their different usage parameters. [...] These parameters contain the individual state of the subject, which uses the information (e.g. its current state of knowledge, its memory capacity, its information processing capacity respectively in general: its intelligence) and situational factors like money (e.g. the availability of time and money for information processing, purpose, organizational background, general culture of information in the current environment). He concludes by defining information with two statements. Information is knowledge in action and information is knowledge in context. These statements are supported by his transformation model of knowledge and information, which is shown in figure

4.2. It shows the central role of context in the course of information administration and information production.

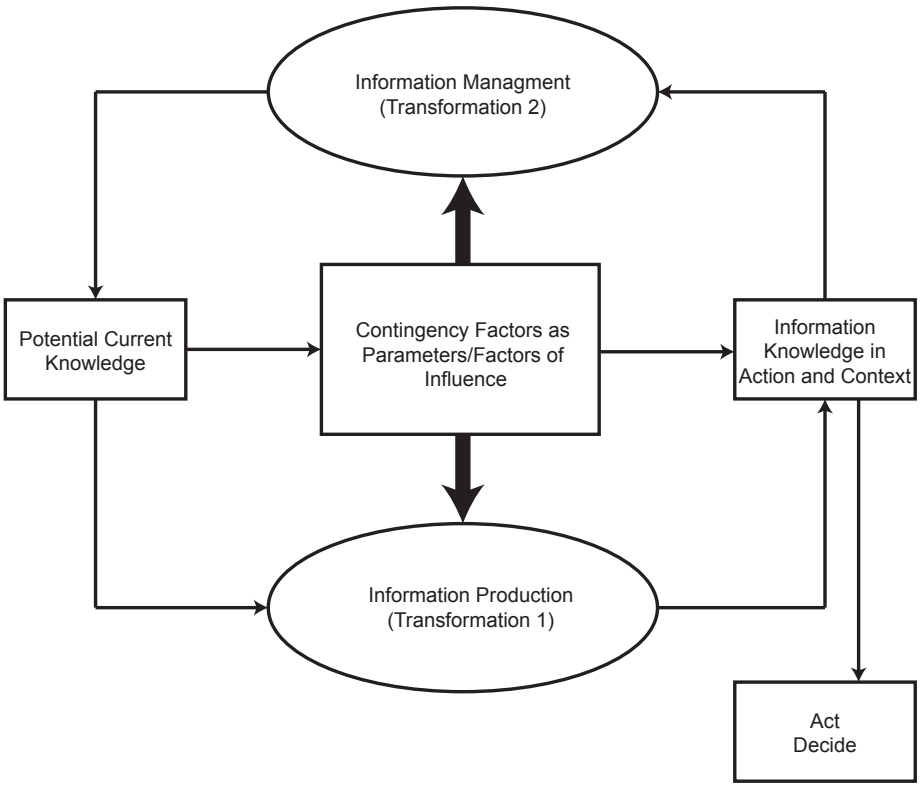


Figure 4.2: Kuhlen's transformation model of knowledge and information. Adapted from Kuhlen et al., 2004, p. 15

Kuhlen's conclusion and his transformation model of information is the basis for the work definition of information for this thesis.

Definition 1. *Information*
Information is produced through a transformation of knowledge in a specific application context to give a usable answer to a specific problem.

4.1.2 Integration-relevant Information

The term integration-relevant information is defined based on the definition of information above. The *context* for this kind of information is the integration phase during the development of component-based embedded automotive software. This includes the execution of software integration and surrounding activities like integration planning or administration. The use or addressed *problem* of this kind of information is to improve the integration in terms of measurability, efficiency and quality.

Definition 2. *Integration-relevant Information*
Integration-relevant information is information with an integration context, i.e., information that is produced during software development of component-based embedded automotive software and is used to improve software integration, integration planning and administration. Examples of integration-relevant information are listed in table 4.1.

Table 4.1: Examples of integration-relevant information available in a typical industry setting. (cf. Schorer, 2010a)

Product Definition	
Product Requirements	Definition of functional requirements of the software product
Product Features	Functional structure of the product
Release Definitions	Assignment of features to specific delivery dates (releases)
Product Basis (opt.)	Existing project as starting point
Product Delta (opt.)	Differences (additions, removals or edits) to the product basis
System Models	
Component Definition	Description of component requirements and functionality, and assignment to product features
System Architecture	Definition of component interactions (leads to dependencies between the components)
Implementations	
Source Files	Component Implementations
Component Descriptions	Description of the component interfaces
Source File Location	Location of the source files (e.g. in the version control system)
Project Annotations	Explicit component delivery and availability dates

4.2 Related Work

There are several supplementary systems for software development and management of software development. There is project management

software, which handles tasks like estimation and planning, scheduling, cost control and budget management and resource allocation. Examples are project management systems, software versioning systems, configuration management systems, ticketing systems, CASE tools, requirement engineering tools or wiki systems. The Software engineering Book of Knowledge (SWEBOK) Bourque and Dupuis, 2004 offers an overview of software engineering tools in section 10. However, there is currently no system or tool that is focused on software integration. There are tools, which cover certain aspects of integration. Some examples are listed in the following. Software design tools capture dependencies of software components, software configuration tools are reconciled for version management and build/release activities. Software development management tools give support for project planning and measurement.

The next section illustrates how integration-related information is collected and used in a typical industry setting and shows the consequences of the missing tool support.

4.3 State of the Art

Collection and use of integration-relevant information during the development of component-based systems is currently not carried out in a structured and effective way in the industry. Interviews with experts uncovered that there exist only few pragmatic approaches to meet the challenges which are caused by the complexity of this particular development step **sotaconti**; Schorer, 2010b. Like many other software products the main focus in its development is on the implementation of new functionalities and improvement of existing functionalities. In component-based architectures this means that the components itself are at a very high quality level concerning the quality of their function. The success of software integration does not primarily require a high quality of the components' functionalities but the knowledge of the components interfaces and their functional interactions, the conformance of these interfaces to the interface definition and a sophisticated integration process.

The first point in time during the development of a software product when an integration-relevant piece of information is generated is when the contract between the customer and the producer of the software product is sealed. They negotiate the requirements and therefore the features of the product. Also schedules for interim releases as well as the date for the final delivery are set. Additionally the project plan determines which resources are used during the product development. This includes resources for integration like integrators or test suites as well. All of these project parameters can change: Features can be removed or

added, resources can be edited, delivery dates can be changed and interim releases can be added or removed. This information is stored and administered through project management tools like Microsoft Project and/or in proprietary project databases, like the LIMAS database. LIMAS is a document management database for software projects, which is used at Continental Automotive GmbH. It includes documents that define the content of the software product and the specifications of the components of the product.

The features of the software product are the starting point for an architectural modeling of the product's software system. Based on the experience of the project manager and his software development team, the software components are reused from similar completed software projects and are edited, extended or replaced to meet the project requirements. The system architecture is often defined on a component view in Matlab Simulink files for functional models. A system-wide description of the architecture is available later in the development process on the implementation layer, e.g., through the AUTOSAR[2] architecture. AUTOSAR is a standardized software architecture for automotive software systems that is developed though the AUTOSAR Consortium, an industry wide collaboration of automotive manufacturers and suppliers.

At Continental Automotive GmbH, the system modeling of the product is also available through the combination of specifications which are available in the LIMAS database and the ADD, a database for component interface descriptions. This kind of system-wide modeling of the product captures the interconnections of the components and makes it possible to determine the dependency relationships between components.

When the system architecture is matured to the implementation level (e.g., the AUTOSAR architecture of the system is complete) the definition of the software component interfaces is the second instance when integration-relevant information is produced. The interconnections of the components lead to functional dependencies between the components, which play a crucial role concerning the stub effort during the integration phase (see section 7.2 for a detailed description). The architectural and the implementation details of the components are combined with the project parameters as described above. This means that each architectural component description belongs to a specific feature of the product and is itself implemented by a set of source files with meta information about their interfaces. The result is the information input that integrators have available. It consists of a selection of source files with meta information and annotated project parameters. The according process is illustrated in figure 4.3.

Figure 4.4 shows a configuration with multiple integrators and soft-

[2] http://www.autosar.org

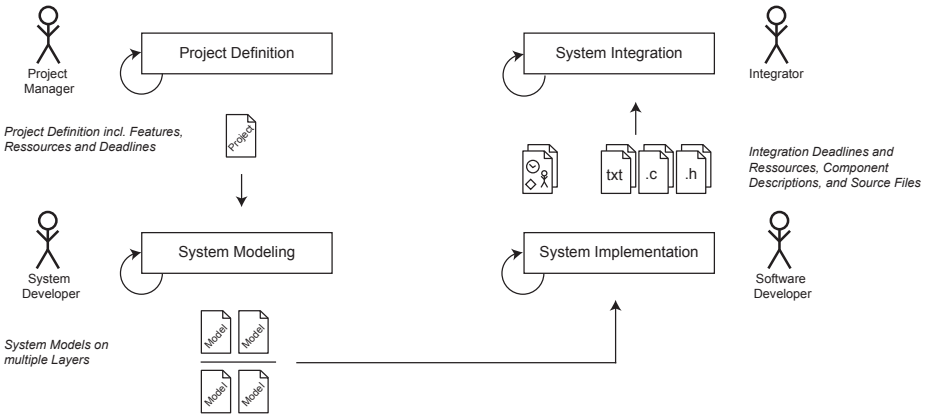


Figure 4.3: Information collection and usage process with a single integrator.

ware developers. Collaboration between integrators and the overall coordination of project's integration activities is achieved through the assignment of time slots in a specific file (commonly a spreadsheet). The assignment of time slots can be coordinated by the project manager but a first-come, first-serve fashion is also common.

A detailed view on the sequence of integration activities is given in figure 4.5. The project description is used to create tickets for integration according to the release date of the component's feature. After the implementation of the component is finished each component has to be integrated before its release date. Figure 4.5 shows an exemplary arrangement of five integration activities. Tickets are assigned for the implementation and integration of *Component E*.

4.4 Analysis

This section inspects the current information collection and usage with respect to four aspects: process, availability, completeness and collaboration. These aspects are used as basis for an analysis of the current challenges.

4.4.1 Process

Concerning the process of the current information collection and usage, the main flaw is the sequence in which integration-relevant decisions are taken. The project manager sets the release dates for product releases and resource allocations for the product at a very early stage of the de-

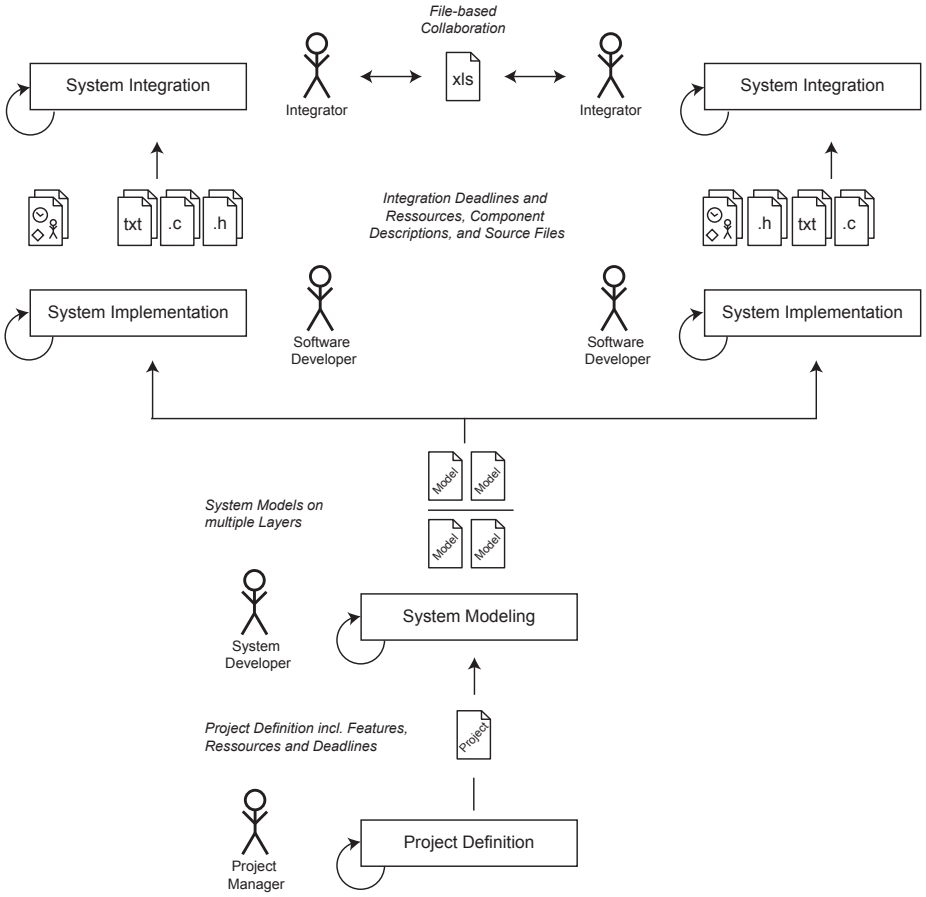


Figure 4.4: Information collection and usage with multiple integrators.

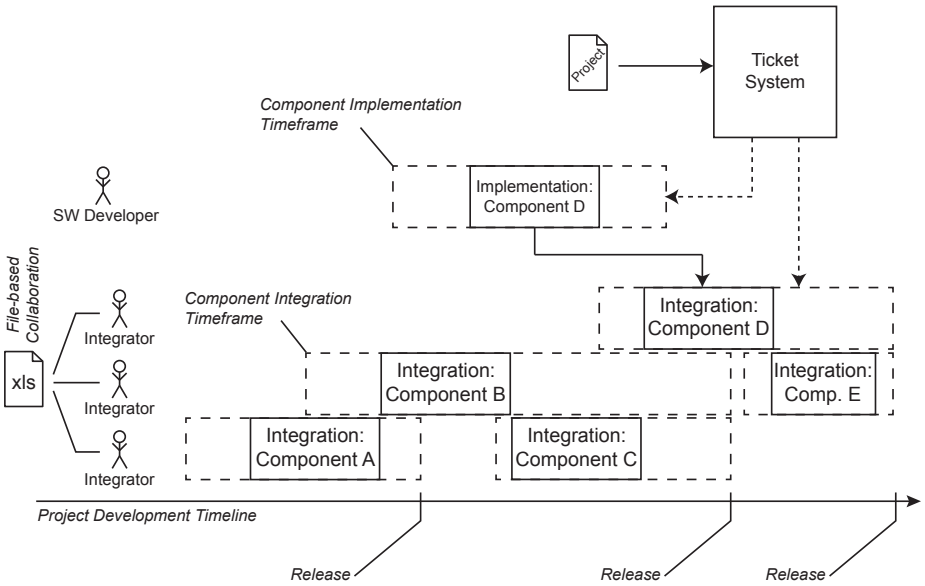


Figure 4.5: Exemplary arrangement of integration activities for five components.

velopment process. Also the system modeling stage, which follows after the project definition, determines the functional decomposition of the product features into components. The most part of integration-relevant information is generated in these two development steps. The results of these decisions are constraints for the integration phase and have a high impact on it at a later point in time in the development time line. The major problem is the lack of understanding of the consequences of these decisions on the course of the integration phase. The information that is generated is not used to perform integration planning or to perform an analysis of the feasibility of the integration process.

The integrators have to perform the necessary integration activities under the aforesaid constraints, on which they have very little influence. They can decide the order of integration activities within these boundaries. The process flaws of the current integration management can be improved through the virtual integration methodology (see chapter 3). Its extension to the current development process introduces mandatory tasks and work products that enable a successful integration planning. The following example illustrates the benefits of a designated integration planning process.

Example The project manager assigns a component A to a release R_0 at a time x_0 during project development and a component B to a release R_2

at a time x_2 with $x_2 > x_0$. Component A has a dependency on component B (e.g., through a direct communication) and the system S at x_0 has only one dependency on component B. The dependency chain would then be $S \leftarrow B \leftarrow A$. The result of the project managers assignment is that the integrator has to develop a component stub for component B a x_0 to perform a complete integration test for S and A. With an integration planning process, which would have made the project manager aware of the additional effort during integration, the project manger would see that switching the release dates of components A and B would reduce integration effort.

4.4.2 Availability

As mentioned in section 4.3, the information that is necessary for integration planning is already present. Integration-relevant information is stored in a variety of places during the development time. Examples are project description files, architecture models, tickets in ticketing systems or component interface definitions. These information sources are not constructed with the purpose of providing information during integration planning or execution. Therefore, the structure of these information sources and their access methods are not sufficient.

Example An integrator is working on a set of components for a particular product. That means that he has integration tickets with integration deadlines for all components in his release. In order to determine a feasible integration plan for his set of components he is required to browse through all available information sources and put together a combined overview of all dependencies and constraints. This task can be time-consuming depending on the quality and accessibility of these documents.

4.4.3 Quality

Since integration-relevant information is stored in information sources that are not focused on integration in particular, the completeness of the information is not guaranteed at all times. That means that even if the integrator is aware of the information sources and is able to access it, there may be information missing to plan the integration for his project. Also, the current information collection and usage does not incorporate quality assurance. That means that there may exist outdated or wrong information about the project and especially the system architecture. This is mainly due to the fact that automotive software development is often functional oriented. The development teams are organized according to

the functionality of the software component and this functionality is documented extensively. The information that is necessary for an effective integration does not concern the functionality directly and is therefore prone to be neglected in the documentation.

Example Assuming that the integrator is aware of his available information sources and is able to access them, there may be information missing or faulty information present. An undocumented input port on a component can lead to an unknown dependency between at least two components. If these components are to be integrated in an order which ignores this dependency, the compilation process will fail unexpectedly and the integration will be delayed significantly since the integrator has to develop an additional component stub.

4.4.4 Collaboration

It is common in the industry that a product is integrated by a group of integrators, which can be members of different development teams or even work at different company locations. The software development and test staff are also required to work together closely with the integrators. Finally there is the project manager, who is responsible for the overall project administration. Assignment of integration activities to the integrators is currently done through the ticket system. These tickets are mainly oriented towards the implementation of the software components and include integration as a follow-up activity. Integration activities and their particular difficulties have to be negotiated between all contributing parties to ensure that the knowledge of each party is efficiently leveraged to determine a feasible integration plan and to execute it subsequently.

The main drawback of current situation is a direct consequence of the lack of collaboration. When the integration is executed by more than one integrator, a low level of collaboration during the planning phase leads to a largely uncoordinated integration phase. In the best case the sequential ordering of the integration activities is determined by the project leader through the assignment of development tickets with explicit integration dates. The worst case is an integration sequence that is not ordered to meet any particular requirements but is instead arbitrarily constructed in a first-come first-serve fashion by the involved integrators. In this case the stub development effort can be very high.

Example Two integrators are each working on a set of components for a particular product. Three components A, B and C of integrator 1 are

dependent on four components D,E,F and G, to be integrated by integrator 2. The dependencies are $A \rightarrow D$, $B \rightarrow E$, $C \rightarrow F$ and $C \rightarrow G$. In a first-come first-serve integration without collaboration the two integrators could produce an integration sequence, that ignores these dependencies (e.g., C, F, G, D, E, A, B). In this case, integrator 1 would need to stub every dependency of his components.

4.5 Information System

The analysis in the previous section showed that major challenges exist concerning information collection and usage for integrators and project managers during the integration phase. As figure 4.1 clearly shows, the integration of component-based software requires activities that are spread over the complete development cycle. These activities are defined in the virtual integration methodology (see chapter 3). There are activities that have a planning character, like compatibility checking and integration planning. Other activities have a more supplementary purpose, like integration monitoring. The activities are also focused on different parts of the software project, the compatibility check will operate exclusively on the software product's architecture whereas integration planning takes into account more project specific environment information. Additionally, the activities have different operation patterns, some are performed in a one time manner and others require a continuous feedback about the current state of the software project.

Despite all the differences of the activities in the virtual integration methodology, the main unifying aspect of them is that they all use information that is already present in the software development cycle, process it and make it accessible to the integrator to enable a more efficient software integration. This characteristic makes it possible to denote this system as an information system. Kunz et al. define the properties of an information system in Kunz and Rittel (1972) as follows (translated from german):

1. Information systems are applications, which shall enable and support the external information of a user (or a class of users) with regard to a class of his (or their) problems.

2. Every information system is someone's information system.

3. A system is an information system, if it contributes to the information, and not because it generates or contains information. Only the data is saved or generated that can be used to inform a user in a given problematic situation.

4. An information system contains only what shall contribute to the external information, i.e., what is not internally generated (like an idea, changes in thinking, etc.). That does not exclude that information systems can trigger internal knowledge-changing processes.

5. An information system is not set up only for a class of actors, but also with regard to a class of problems, whose solution it shall support.

6. Every information system is unique.

Information System for Software Integration

The definition of Kunz and Rittel, 1972 applies fully to the designated application as tooling support for the virtual integration. The theoretical background and literature sources on this topic can be found in the field of business or management information system studies. This Field offers concepts and a terminology that can be transferred to the challenges during the integration phase and the requirements for the tooling support of the integration activities.

The *Integration Information System* (IIS) supports integrators before and during the integration with supplementary information (cf. 1 and 2 above). The IIS processes information that is already present during the development process. This data is then used to support the user during the integration phase. Integration plans and administrative data for integration are generated in the IIS on the basis of the already existing data and the project environment (cf. 3 and 4 above). The IIS is a specifically designed information system for one problem field, the effective management and execution of software integration, and for a specific class of users, the integrators and the corresponding project managers (cf. 5 and 6 above). The information system targets software integrators as main user group. They are responsible for integration planning, execution and integration test. The project manager provides the general parameters of the project like the projects interim release dates. The user roles are discussed in detail in chapter .

Definition 3. *According to Panyr in Panyr, 1986, an information system is formally defined as a 7-tuple IS = (A, W, Q, I, E, U, D), with*

- *A: Input function to construct internal representation (access function, learning function)*
- *W: Internal representations (document set, knowledge base)*
- *Q: Input set as set of all permitted input configurations (problem formulation, search term)*

- *I: Output function (inference function, retrieval function, ranking function)*
- *E: Output set as set of all possible output configurations (problem solution, system recommendation)*
- *U: Update function of the internal representations (learning function, relevance feedback)*
- *D: Dialog component, interface*

Panyr's definition conforms to the properties of an information system as defined by Kunz & Rittel. It has a very distinct separation between the internal knowledge of the information system and the external information, e.g., the input and output sets. However, Panyr defines an information system more extensive than Kunz & Rittel. He introduces the basic components and defines their functions. His definition can be seen as an implementation of the requirements that Kunz & Rittel defined.

This definition is used later in the definition of the IIS, when for all elements of the tuple the corresponding parts in the IIS are identified (see section 4.6).

4.6 Definition of the IIS

In the following, the internal representation W, the input set Q and output set E are described for the IIS according to the Panyr's definition of an information system (cf. definition 3).

4.6.1 Input Set

The input set Q for the information system is divided into two information sources, the software architecture of the system and the properties of the software project.

Input 1: Software Architecture The software architecture can be available in different levels of abstraction. In the case of automotive software development the system architecture is described in models of the software system or sets of models for specific subsystems. These models can be functional as well as architectural descriptions of the software system. Examples are EAST-ADL or EAST-ADL2 system models Cuenot et al., 2007; Lönn et al., 2008 which model all abstraction levels of the software product, MATLAB Simulink models that are used to design complex controls MathWorks, Inc., 2011, AUTOSAR system descriptions AUTOSAR Administration, 2008 or other proprietary descriptions of the software system (e.g., through source code re-engineering).

These forms of description are very distinct in syntax and application case. However, they share the common concept of objects that provide a functionality and connections between them, through which they exchange some kind of data. Secondly, they are implemented through some kind of source code at a point in time in the development process. These source files are the objects that need to be integrated to form a working version of the software product and are therefore the main objects of interest. The interconnections between the model objects imply a dependency between the provider of the data and its consumer. This dependency is our second point of interest in the system models since it determines a notion of order between the objects (e.g., the dependent object will not be fully functional if the providing object is not yet available).

Figures 4.6, 4.7 and 4.6 show typical architectures of automotive applications in MATLAB Simulink, EAST-ADL2 and AUTOSAR representation. The common design of interconnected and therefore dependent components is visible in all three illustrations.

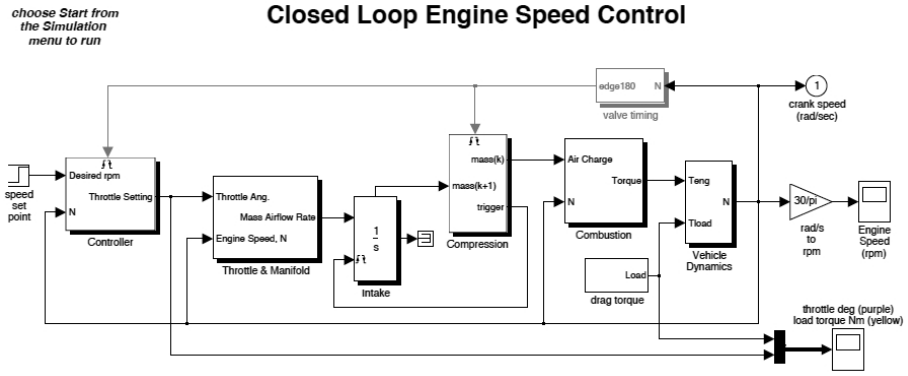


Figure 4.6: MATLAB Simulink example: An engine model with a discrete-time PI controller to regulate speed. MathWorks, Inc., 2000

Input: Software Project Definition The second input for the IIS is the description of the software project. It is most commonly defined with project management software like MS Project. The project description includes among others resource management , risk management, cost management, time management and scheduling and task management for every part of the final product. For the IIS, the relevant parts are the delivery deadlines for every software component (i.e. its release), the end of implementation and the end of module test respectively and the availability and cost of resources (e.g., the integrators).

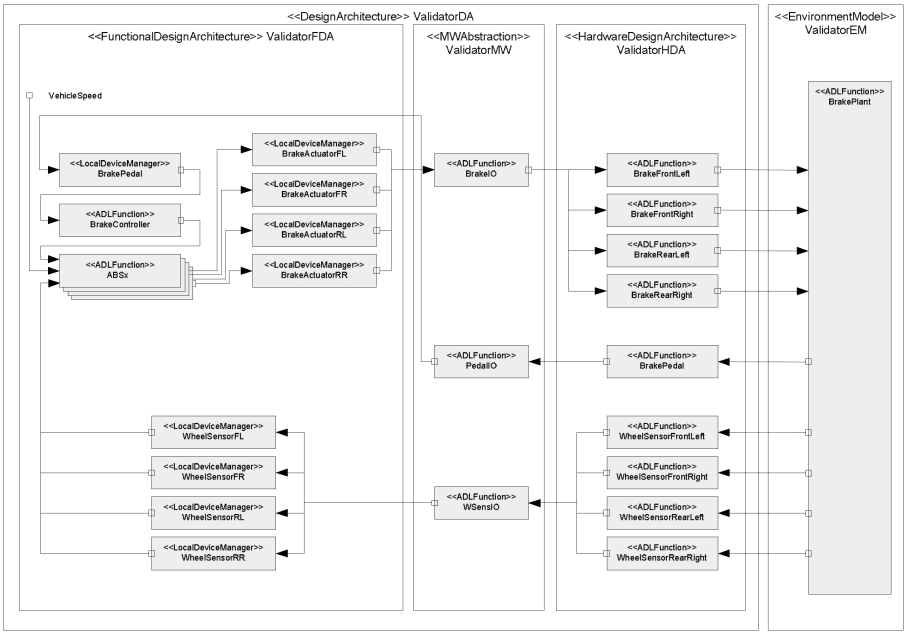


Figure 4.7: EAST-ADL2 example: Design architecture of a brake system validator, including middleware abstraction, hardware architecture and environmental model. Stappert et al., 2010

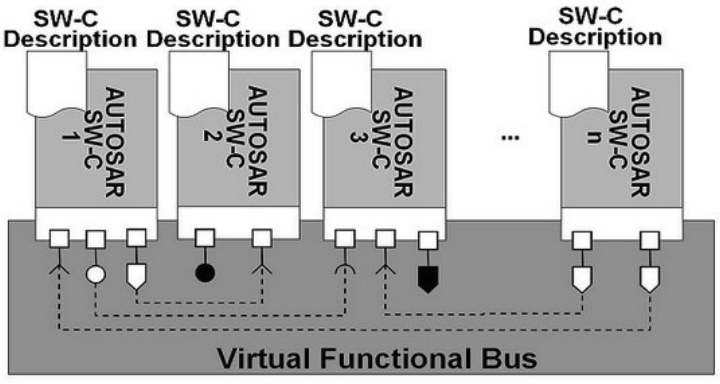


Figure 4.8: AUTOSAR example: The virtual function bus connects several software components. Sandmann and Schlosser, 2008

4.6.2 Input Function

The input function A in the case of the IIS is a parsing or transformation facility that produces the internal representation of the information system from the input set Q. This mapping will extract the integration-relevant information from the system architecture and software project descriptions.

4.6.3 Internal Representation

The internal representation *W* (cf. definition 3) combines the system architecture and the project description into a general model. That means that an abstract model of the system architecture, consisting of a set of abstraction layers with interconnected components, is annotated with the necessary project specific attributes, e.g., component release dates. Only integration-relevant information (cf. definition 2) of the final software product is preserved in the internal representation of the IIS.

4.6.4 Output Function

The output function I (or inference function) uses the internal representation W to give a response to the user's problem (constructing the output set *E*). In the case of the IIS, the information has multiple output functions. The main function is to give the user a proposal for a valid integration sequence under the given constraints from the internal representation of the system architecture and project specification. Furthermore, there are output functions that support the user during integration by giving them additional information during the integration phase.

4.6.5 Output Set

The output set *E* of the IIS is a selection of valid integration plans for the particular system architecture under the project's constraints. It also contains supplementary information for the integrator, like integration sequence visualization, cost measurements and administrative information. In other words, the result is an assignment of the product's components to integration time slots during the integration time frame and additional information for manual integration planning and administration.

4.6.6 Update Function

The update function U is a refinement of the IIS's integration model. This may be necessary through missing information in the initial input set or

changes that occurred in the course of the software development.

4.6.7 Dialog Component

The dialog component D of the IIS is a graphical user interface, which is embedded in an Eclipse[3] plug-in. It consists of tabbed UI element with tabs corresponding to each aspect of the information system, e.g., there is a separate tab for integration monitoring and another tab for integration planning.

4.7 Classification

This section classifies the class of information system to which the IIS belongs. The definition of the according information system class is applied to the IIS and the feasibility of the requirements is evaluated for the particular class of information system.

4.7.1 Decision Support System

Decision support systems (DSS) were defined initially under the term management decision system by Scott-Morton in Scott Morton, 1971. He describes them as "interactive computer-based systems, which help decision makers utilize *data* and *models* to solve unstructured problems". Later, Keen and Scott-Morton defined a DSS as follows Keen and Morton, 1978:

> Decision support systems couple the intellectual resources of individuals with the capabilities of the computer to improve the quality of decisions. It is a computer-based support system for management decision makers who deal with semi-structured problems.

Alter defined a taxonomy of DSS in Alter, 1975 which is mainly driven by the DSS's technological aspects. Table 4.2 gives an overview of this taxonomy.

In the requirements for the IIS, we can identify two requirements that fall into the class a DSS: *Integration Plan Measurement* and *Integration Planning Support*. The DSS-part of the IIS can be classified according to Alter's DSS taxonomy as an optimization model-based DSS. The complex nature if the software architecture and the project environment make it an elaborate model. Since the goal is to propose the best possible solution

[3] www.eclipse.org

Table 4.2: Taxonomy of Decision Support Systems according to Alter, 1975

Data-oriented	
File Drawer Systems	Allow immediate access to data items
Data Analysis Systems	Allow manipulation of data by tailored or general operators
Analysis Information Systems	Provide access to a series of databases and small models
Model-oriented	
Accounting Models	Calculate the consequences of planned actions using accounting definitions
Representational Models	Estimate the consequences of actions without using or partially using accounting definitions
Optimization Models	Provide guidelines for action by generating an optimal solution
Suggestion Models	Provide processing support for a suggested decision for a relatively structured task

in a very large problem space the DSS can be referred to as an Optimization Model DSS.

A detailed definition of a decision support system and its properties can be found in Holsapple, Whinston, Benamati, and Kearns, 1996. These are illustrated in figure 4.9 and can be described by the following five properties:

1. A DSS includes a body of knowledge that describes some aspects of the decision-maker's world, that specifies how to accomplish various tasks, that indicates what conclusions are valid in various circumstances, and so forth.
2. A DSS has an ability to acquire and maintain descriptive knowledge (i.e., record keeping) and other kinds of knowledge as well (i.e., procedure keeping, rule keeping, etc.).
3. A DSS has an ability to present knowledge on an ad hoc basis in various customized ways as well as in standard reports.
4. A DSS has an ability to select any desired subset of stored knowledge for either presentation or deriving new knowledge in the course of problem recognition and/or problem solving.
5. A DSS can interact directly with a decision maker or a participant in a decision in such a way that the user has a flexible choice and sequence of knowledge management activities.

They define, that the architecture of a decision support system consists of the following four parts:

- Language System (LS)
- Presentation System (PS)
- Knowledge System (KS)
- Problem-Processing System (PPS)

The language system consists of all messages the DSS can accept. The presentation system consists of all messages the DSS can emit. The knowledge system consists of all knowledge the DSS has stored and retained. By themselves, these three kinds of systems can do nothing, neither individually nor in tandem. They are inanimate. They simply represent knowledge, either in the sense of messages that can be passed or representations that have been accumulated for possible future processing. Although they are merely systems of representation, the KS, LS,

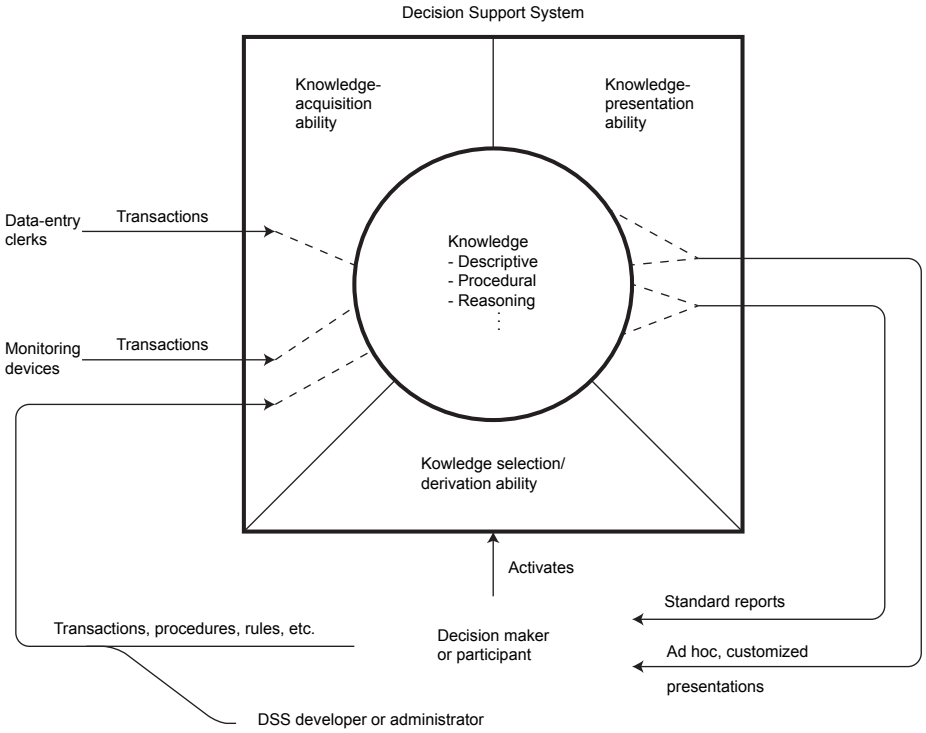


Figure 4.9: Typical Decision Support System. Adapted from Holsapple et al., 1996, p.144.

> and PS are essential elements of a DSS. Each is used by the fourth element: the problem-processing system. This system is the active component of a DSS. A problem-processing system is the DSS's software engine. As its name suggests, a PPS is what tries to recognize and solve problems (i. e., process problems) during the making of a decision.Burstein and Holsapple, 2008

According to Holsapple and Whinston Holsapple et al., 1996, decision support systems can be classified into six architectural classes: Text-oriented DSS, Database-oriented DSS, Spreadsheet-oriented DSS, Solver-oriented DSS, Rule-oriented DSS, and Compound DSS. The first five classes are very distinct in the type of data stored in their knowledge system and need therefore specialized language, presentation and problem-processing systems to access this knowledge. A combination of different types of decision systems is called a compound decision support system.

In Holsapple and Whinston's categorization, the IIS complies with the solver-oriented DSS. This is especially true for the *Integration Planning Support* of the IIS. They define a solver-oriented DSS as a decision support system that contains a specialized solver (a procedure or algorithm) that is embedded within the DSS. There exist two types of solver-oriented DSS, a fixed and a flexible solver-oriented DSS. The flexible solver-oriented DSS enables the user to choose from a set of solver modules and combine them according to his specific purpose. Since the IIS does not have a need for such a mechanism the fixed solver-oriented DSS, which incorporates only one designated solver instance, is sufficient. Figure 4.10 shows the structure of a fixed solver-oriented DSS.

However, there are requirements, which cannot be fulfilled with such a solver-oriented approach. Integration-relevant information bundling, Compatibility Identification and Visualization, Integration Plan Measurement, Integration Planning Support, Integration Administration, Integration Status Checking and Integration Monitoring are requirements that are supplementary for the integrators decisions in the integration phase. These requirements can be satisfied by implementing a database-oriented DSS.

Holsapple defines a database-oriented DSS in contrast to a text-oriented DSS as follows. He states that, rather than treating data as streams of text, they (the data tables) are organized in a highly structured, tabular fashion. The processing of these data tables is designed to take advantage of their high degree of structure. It is, therefore, more intricate than text processing.

The form of data storage that is used in the IIS is not exactly a database. But the model-based descriptions of the software product, which form the knowledge base for the DSS, are comparably structured. The database

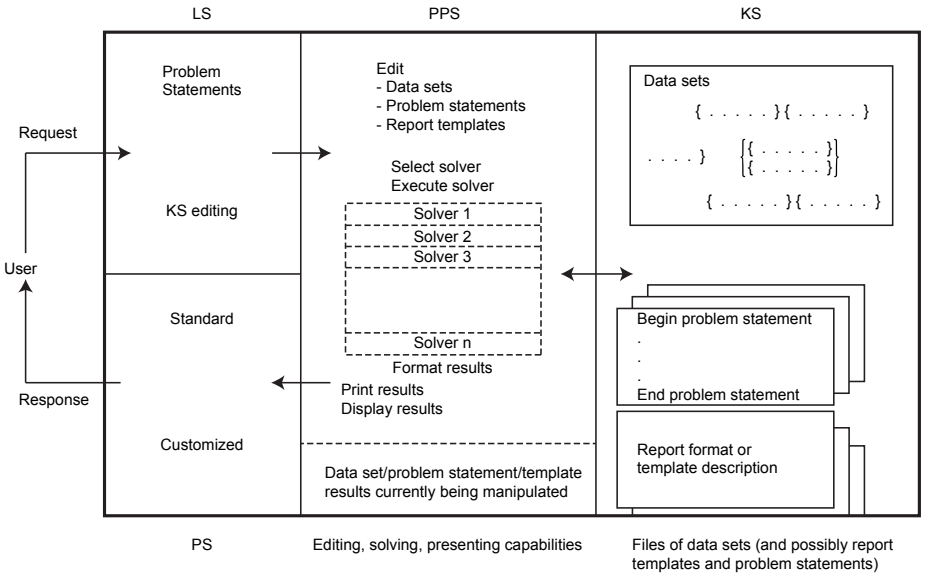


Figure 4.10: The structure of a fixed solver-oriented decision support system (Holsapple et al., 1996).

control system of a database-oriented DSS has its equivalent in the model API that provides access to the model elements and attributes, enable storing and loading of information into models and additional functionalities such as validation. The database-oriented DSS's custom-built processing system is provided in the IIS through a set of output functionalities (e.g., Eclipse UI elements), which give results to predefined requests. A detailed description is provided in section 5.3.

In order to bring these two types of decision support systems together we can leverage Holsapple's definition of a compound decision support system, which is illustrated in figure 4.11. The IIS represents a so-called synergetic approach to a compound DSS (cf. Burstein and Holsapple, 2008, p. 181). In the synergistic approach to integrating traditionally distinct knowledge management techniques, there is no nesting, no dominant technique, and no secondary nested techniques. All techniques are integrated into a single tool that allows any capability to be used independently of another, or together with another within a single operation Burstein and Holsapple, 2008, p. 181.

4.7.2 Application of Holsapple's Architecture

Figure 4.12 shows how the Holsapple's definition of a compound decision support system is applied to the IIS. As described before, the IIS is a compound decision support system, which contains database and solver

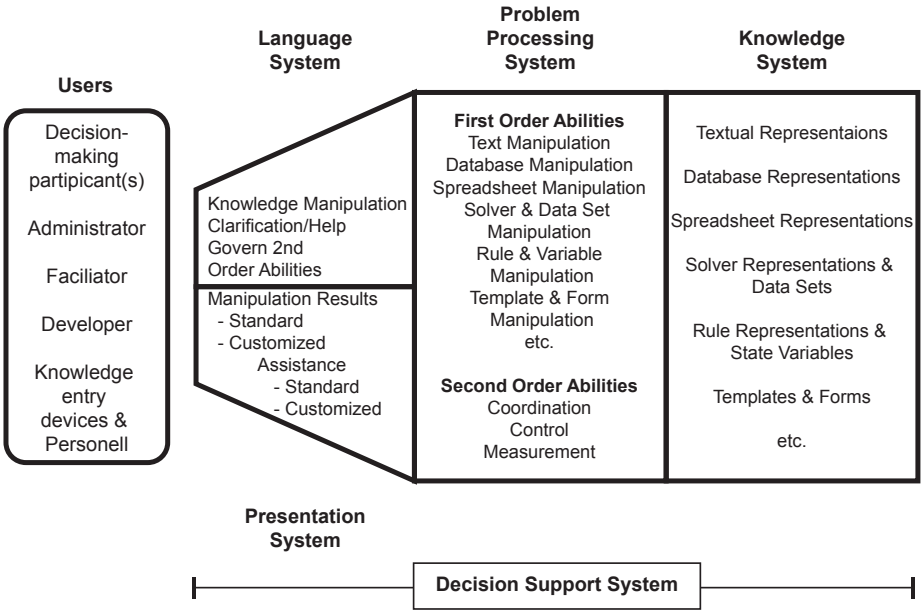


Figure 4.11: The structure of a compound synergetic decision support system Burstein and Holsapple, 2008.

oriented aspects. The two concepts are integrated in a synergetic fashion to provide the user with a single system throughout the whole integration phase.

The knowledge systems contains two types of knowledge in the IIS — the integration model, which is the basis for all database-oriented decision support, and additional solver-oriented representations that are used to generate responses through the integration planning system. The problem-processing system has therefore two types of abilities that are designed according to the requests and responses through the language and presentation system respectively.

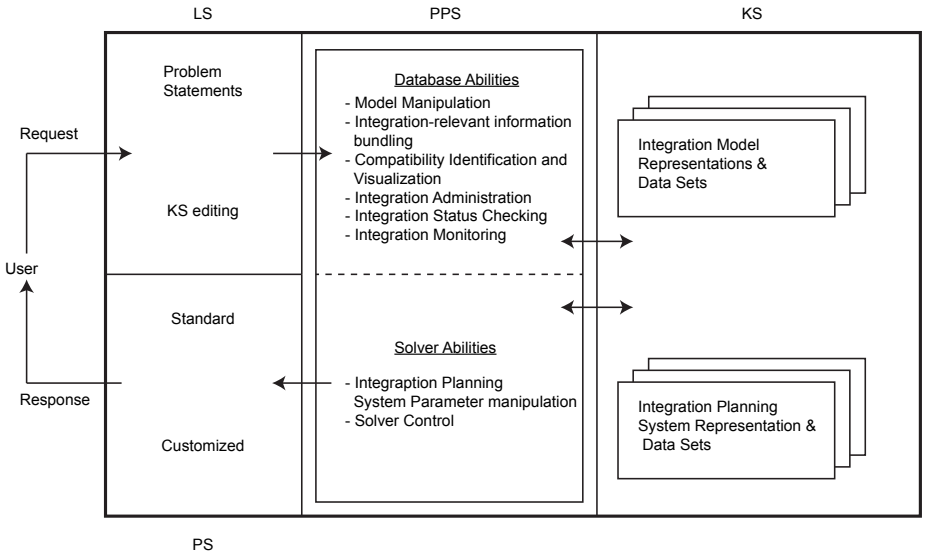


Figure 4.12: The structure of the IIS modeled as a compound synergetic decision support system.

Chapter 5

IIS Prototype Architecture

THIS section defines a prototype architecture for the integration information system. It includes a definition of functional and non-functional requirements for such an information system and defines its parts and their interconnections in detail.

5.1 IIS Requirements

The implementation of an information system in the context of software integration in the automotive domain has to fulfill the following functional and non-functional requirements (cf. Schorer, 2011b). The requirements were defined through workshops with industry experts according to the techniques described in Pohl, 2008. The integration state-of-the-art survey (cf. section 1.1.2) provided additional insights on the requirements.

5.1.1 Functional Requirements

Integration-relevant information bundling The IIS shall mainly provide means to give the integrators access to an abstracted model of the system and which is designed to emphasize properties, which are necessary to perform an effective integration. This means the information system internal model focuses on these vital parts of the system and masks out parts, that are irrelevant for the integration.

Compatibility Identification and Visualization The information system will use the available data to identify incompatibilities between components and make them visible to the integrator.

Convenient Integration Planning Additionally, the integration information system shall support the integrators in their decision on the choice of a feasible integration plan. Therefore, the integration sequences need to be visualized in a way that they can be evaluated even with an incomplete knowledge of the systems properties.

Integration Plan Measurement The quality of the integration plan shall be measured with system of metrics that capture the individual costs of the integration plan. This metric shall be configurable to fit the projects individual properties.

Integration Planning Support The user shall be supported by an intelligent support mechanism for integration plans. This mechanism shall propose an integration plan that is calculated with use of the aforesaid configurable metric for integration plans.

Integration Administration Such an information system shall provide an effective work management for integrators and organize the integrators' tasks according to project deadlines and priorities.

Integration Status Checking The information system shall provide a verification mechanism to check the prerequisites for each aforesaid task. This includes mandatory model properties as well as communication to the configuration management system to check the availability of required source files.

Integration Monitoring As final purpose, the information system is supposed to cover the integration phase in a more general kind. The integration phase of the projects inside of single organizational unit, e.g., a working group, a department, a company division, are continuously under investigation to discover potential gradual quality losses. This aspect can be seen as a continuous integration monitoring.

5.1.2 Non-Functional Requirements

Eclipse Integration The information system shall be implemented as plug-in in the Eclipse workbench. This shall enable a seamless integration with other tools since Eclipse is a widespread tooling platform in the automotive domain.

Software Abstraction Layer Independence The information system shall be able to operate with software architectures on different abstraction layers as input models. This requirement is conform to the generality of the virtual integration methodology, which covers the left side of the V-model (i.e., different levels of abstraction of the software product's architecture).

5.2 Overview

Figure 5.1 shows the conceptual overview of the information system and its environment. The information system incorporates data import functionalities, the connection to the software build system and the version management system, the interface to the integration planning system, the model storage and the user interfaces to integrators and project managers. The following section will describe the building blocks of the IIS and how they are interconnected to enable support during all virtual integration activities.

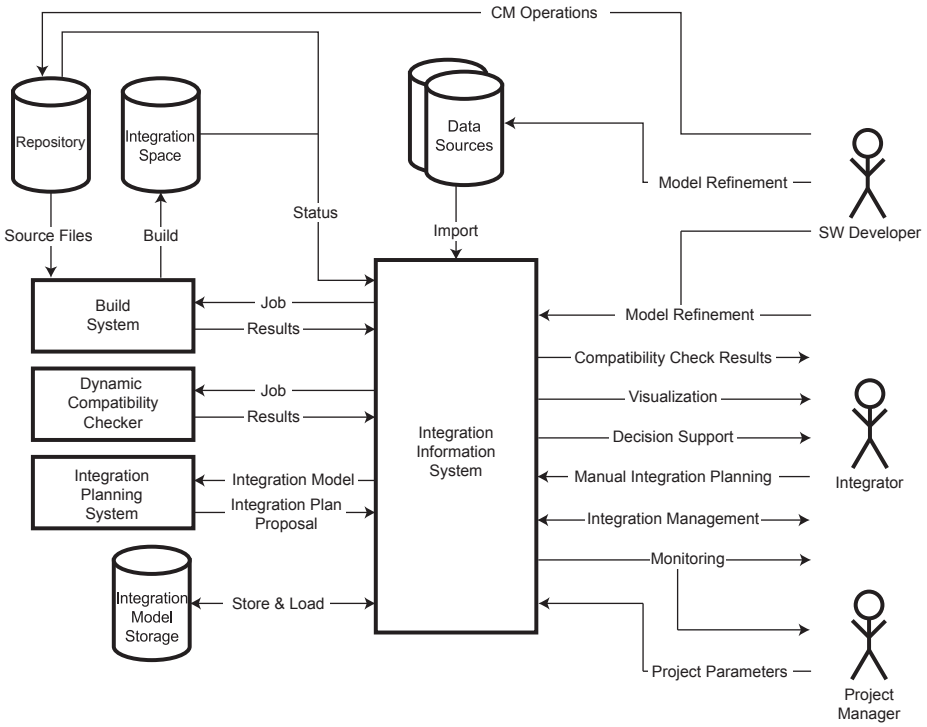


Figure 5.1: Concept of the Integration Information System

Core Components The *Integration Information System* provides the integrators and project managers with access to integration-relevant information. It is connected to an *Integration Model Storage*, that is the main repository for integration-relevant information for all product developments. It also offers a front end for integration monitoring and management, compatibility check, integration visualization and integration planning. The *Integration Planning System* manages computer-aided integration planning mechanisms. The execution of the integration sequences, which were designed in the IIS, is performed by the *Build System*. The *Dynamic Compatibility Checker* is also a separate subsystem that is connected similar to the build system.

Roles There are three main roles in interaction with the IIS. The *Project Manager* contributes the project specific information of the product. He is also able to monitor the current integration status. The *SW Developer* stores the current status of his component regarding integration in the IIS. The *Integrator* is the central role for the IIS. He is responsible for the maintenance of the integration model, compatibility checking, integration monitoring and integration sequence visualization. Furthermore, he uses the IIS for manual integration planning with support through the decision support mechanisms of the attached integration planning system. These roles are described in detail in section 2.1.

Environment The environment of the IIS consists of three repository systems. There are *Data Sources* for the import of integration-relevant data into the IIS. Examples for these data sources are architecture modeling systems, requirement modeling systems or project management systems. Attached to the build system is the Configuration Management System (*CM System*), where the component implementations (source files) are stored. Parallel to the CM system is the *Integration Space* where the builds for each integration stage are stored.

5.3 Integration Information System

The parts of the integration system information are illustrated in figure 5.2. The front end of the IIS consists of the *Model Manager*, the *Editor*, the *Visualization* and the *Integration Administration and Monitoring*. Internal subsystems of the IIS are the *Build System Adapter*, the *Compatibility Checking Adapter*, the *Integration Planning System Adapter* and the *Model Import*.

Model Manager The model manager is the front end to the data import

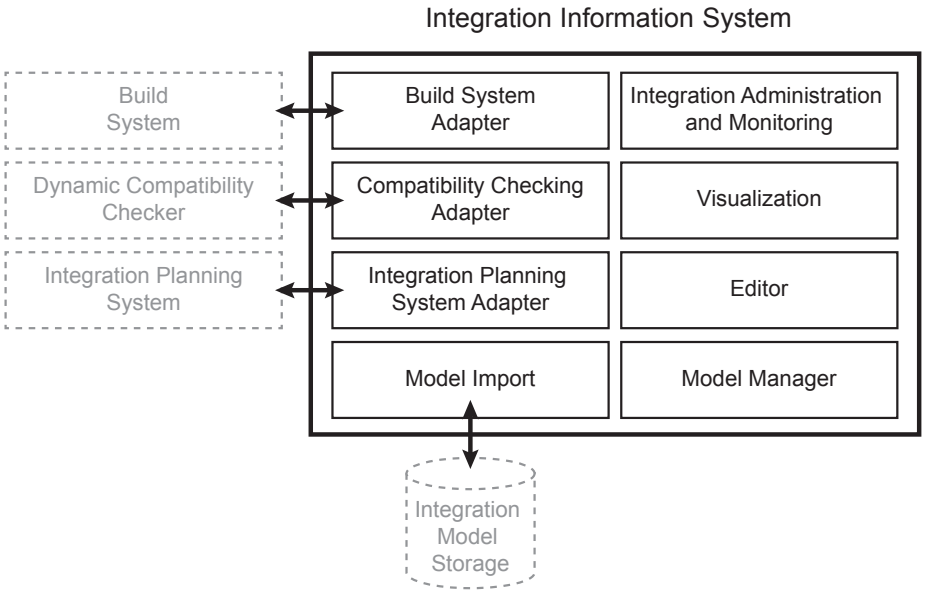


Figure 5.2: Elements of the IIS and the adjoining systems.

facilities of the IIS. It also supports revisioning of integration modules through a designated user interface.

Model Import The IIS model import as described in section 5.4

Editor The tree editor is used to edit all aspects of the integration model. The user can add or delete model elements and change their attributes.

Visualization The integration sequence visualization provides users with a simple method to evaluate integration sequences and to identify potential problems in these sequences. This subsystem is explained in detail in section 5.9.

Integration Administration and Monitoring The integration administration is a user interface to manage tickets and resources for the integration. Integrator can create, edit and mark integration tickets as resolved. Integration monitoring is a comparison and evaluation tool that gives integrators and project managers an overview of completed integration projects and compare them with the current integration project. The two modules are described in detail in section 5.5 and 5.6.

Build System Adapter The build system adapter is the front end to the build system. It is used to create and edit the build descriptions, to launch builds and to give information about the build status and

result. The build system adapter is described in detail in section 5.10.

Compatibility Checking Adapter The compatibility checking adapter is the control interface for the external dynamic compatibility checker, which is described in section 5.7.2. This includes editing support for the required model additions, verification control and result evaluation.

Integration Planning System Adapter The integration planning system adapter connects the IIS to the integration planning system. It includes a configurable generator for the integration system's file format, a planning job control interface as well as a result presentation. The adapter is described in detail in section 5.8 along with the implementation of the integration planning system.

5.4 Model Manager

One of the key points of the virtual integration methodology is the aim to incorporate all integration relevant aspects of a software product in every stage of development. These information sources are described in detail in chapter 3 and in section 4.6.1. The integration information system incorporates an import of AUTOSAR models. The result of this import is an instance of the virtual integration meta model.

Eclipse Modeling Framework The meta model is implemented in the IIS with use of the Eclipse Modeling Framework (EMF)[1]. The EMF project is a modeling framework and code generation facility for building tools and other applications based on a structured data model. From a model specification described in XMI, EMF provides tools and runtime support to produce a set of Java classes for the model, along with a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor Eclipse Foundation, 2012b. It consists of three basic parts (cf. Steinberg, 2008):

- Core Runtime
  - Notification framework
  - Ecore meta model
  - Persistence (XML/XMI), validation, change model
- EMF.Edit

[1]Eclipse Modeling Framework: www.eclipse.org/modeling/emf/

  - Support for model-based editors and viewers
  - Default reflective editor

- Codegen
  - Code generator for application models and editors
  - Extensible model importer/exporter framework

EMF enables an easy development of model-based editors and provides editing, validation and persistence support for these editors. The models, which are described with use of EMF are instances of EMF's integrated meta model Ecore. Ecore itself is a class-based implementation of the OMG's Essential Meta Object Facility (EMOF) standard (EMOF is a partial implementation of the MOF 2.0 standard. It is a subset of CMOF, the Complete MOF.) Object Management Group, 2012.

5.4.1 Autosar Import via Model Transformation

Artop (Autosar Tooling Platform) provides means to develop AUTOSAR modeling plug-ins for Eclipse Rudorfer, Voget, and Eberle, 2010. Therefore, it provides an ECore implementation of the AUTOSAR meta model and additional utility functionalities. Ecore is the underlying meta model of EMF. Since the virtual integration meta model is also based on the ECore meta model from the Eclipse Modeling Framework, it is possible to introduce a model to model transformation between instances of these two meta models. A model transformation is a technique in the field of model driven engineering, which makes it possible to implement a reliable and automatable conversion of models. Model transformations can be carried out between models that are conform to the same meta model (endogenous transformation) as well as models of different meta models (exogenous transformation). Model transformations can be defined as unidirectional, i.e., with one input model and one output model. A bidirectional model transformation can operate in two directions. Both models can be input or output for the transformation. Since a model transformation is always decried and executed in conformance to the involved meta model it ensures a high integrity of the transformation results. The OMG defined a standard for model transformations named QVT (Queries, Views, Transformations) Object Management Group, 2011. The QVT language itself is an extension of the OMG's OCL (Object Constraint Language). The Eclipse M2M (Model to Model) Eclipse Foundation, 2012a project provides an implementation of the OMG's standard. This implementation can be used by including the Operational QVT plug-in in the Eclipse workbench.

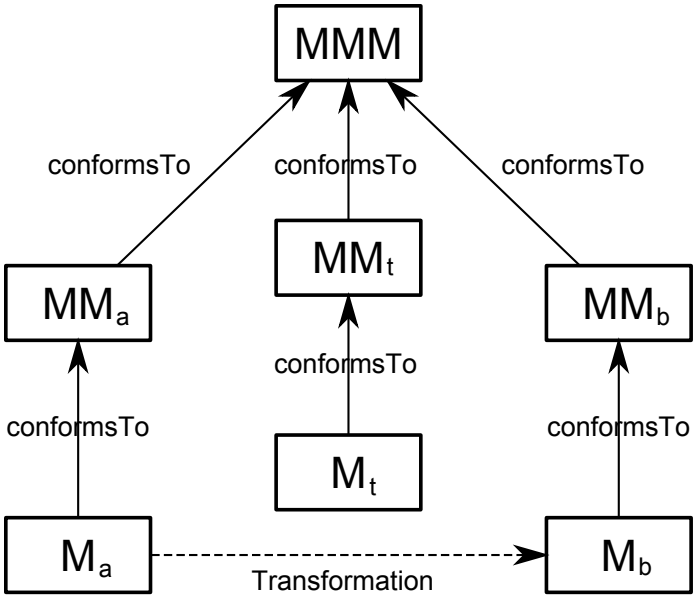


Figure 5.3: Schematic illustration of a model transformation. Abbreviations and application examples for the IIS are listed in table 5.1.

Table 5.1: Description of model transformation elements and examples in the IIS according to figure 5.3.

Abbrev.	Description
MMM (meta meta model)	ECore (Eclipse Modeling Framework meta model)
MMa (meta model)	Autosar meta model
Ma (model)	Autosar model
MMt	Model transformation language
Mt	Model transformation code
MMb	IIS meta model
Mb	IIS model

Since the AUTOSAR standard is available in Artop in six different releases, namely 1.0, 2.0, 2.1, 3.0, 3.1 and 4.0, model transformation for all of these revisions were incorporated in the IIS.

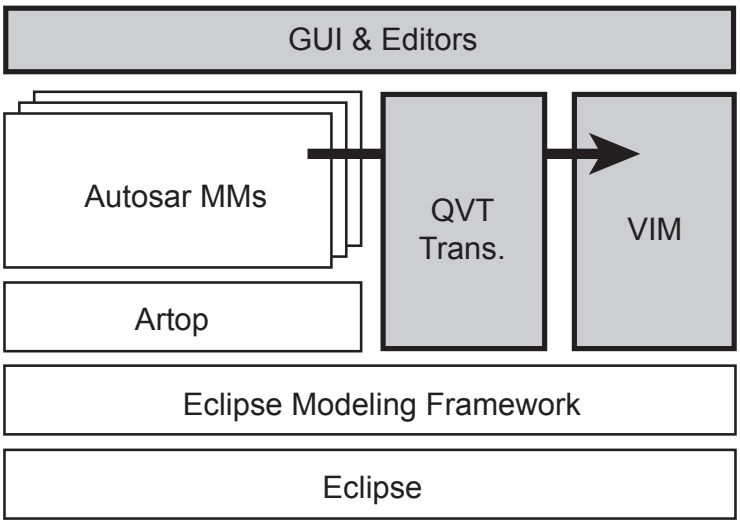


Figure 5.4: Architectural view on the model transformation for Autosar import into the IIS. Parts, marked in grey were newly developed.

The architectural concept of the model transformation with QVT is illustrated in figure 5.4. The eclipse platform and the EMF layer form the foundation for the model transformation in the IIS. The Autosar meta models are located on top of the Artop framework, which encompasses implementations of AUTOSAR meta model releases and a number of related services including AUTOSAR XML Schema Definition (XSD) compliant serialization, rule-based validation, tree and form-based views and editing, and template-based target code, documentation and report generation, and more Artop User Group, 2012. Also on top of the EMF layer is the QVT model transformation and the IIS meta model. The transformation is triggered trough the Model Manager in the IIS.

Other Data Import Mechanisms The import of AUTOSAR system models described above shows how the IIS can be embedded in an AUTOSAR system modeling tool chain. A very similar approach can be used to import EAST-ADL models into the IIS. The meta model of EAST-ADL is available as Ecore model though the Papyrus UML modeling platform CEA LIST, 2012; ATESST2, 2012. This meta model can be used to implement a model transformation like the one described for the AUTOSAR models. Since Matlab Simulink is a proprietary piece of software, there is no official released meta model available. There are projects that im-

port Simulink files through parsing, like ConQAT TUM, 2012 or the kth-simulink-exchange ATESST2 and KTH, 2011. These libraries can be used and extended to provide an import mechanism for EAST-ADL and Simulink.

Import of project definitions can also be established through parsers like the MPXJ Project, 2011. MPXJ is able to read Microsoft Project .mpx files. The parsing results can then be used to generate according model elements in the IIS model.

All the above mentioned import mechanisms need a project to architecture mapping to connect the system architecture with the project information. This can be done interactively during the import process.

5.4.2 Model Manager

In addition to the data import mechanisms for an initial creation of an integration project, the model manager features an import functionality for existing integration projects. This enables reuse of existing, already imported system architecture descriptions and especially complete or partial integration schedules. The feasibility of these schedules has been confirmed in practical use. Therefore, it is possible to adapt them for new integration projects with similar project environments and/or similar system architectures.

The model manager enables also a model-based revisioning of the integration project. Revisions of system architecture descriptions and integration schedules can be saved inside of an integration project.

5.5 Integration Management

Integration management is used to prepare and later for the actual execution of the integration plan. It requires a dedicated integration ticket system to enable coordination during the integration phase. Figure 5.5 shows how integration management is carried out in practice.

The IIS project is built for the development project. The integration plan is defined with use of the project specific information (e.g. release dates or available resources) and the system architecture (see section 5.8 for a detailed description). With the integration plan it is then possible to assign the integration tickets for every ticket to the integrators. The integration ticket changes its status from *assigned* to *ready to integrate* when the component test for the particular component is passed successfully. In ticket system, the ticket is set to active and all shared resource for this ticket are reserved. Other tickets with identical resource sets can not be integrated while the ticket's status is active. When the integration is completed and the integration test is finished, the shared resources are

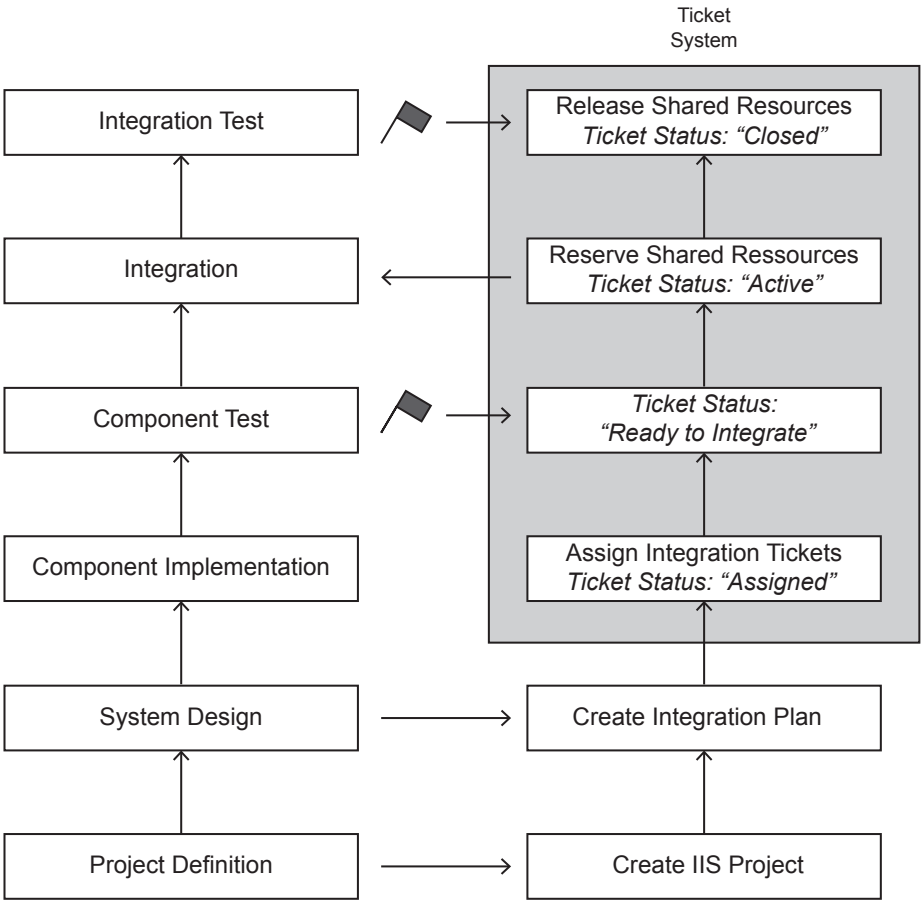


Figure 5.5: Integration Management Workflow.

released and the ticket's status is changed to *closed*.

5.6 Monitoring

The integration monitoring module enables users to get an aggregated view on the integration phases of different projects. Since the projects can share a similar system architecture and organizational properties, this comparison can be very insightful for the project.

The monitoring serves two purposes. The first one is to evaluate the integration quality over time and the second one is to enable a comparison between the current integration project and completed projects from the past. The source projects for this comparison can either be in the standard workspace of the IIS or loaded from a CDO model repository (cf. section 5.4.2).

The combination of a configurable list viewer and a set of charts provide a convenient way to compare integration projects. The list view displays key properties of the projects, like the number of components, number of integration steps and the sum of integration costs. The charts give a more detailed view on a single aspect of the selected projects, e.g., a comparison of the integration cost with the complexity of the system architecture of the projects.

5.7 Compatibility Checking

Compatibility checking between component models requires separate implementations for static and behavioral (and dynamic) compatibility.

5.7.1 Static Compatibility

As described in section 2.3, static compatibility means that the interface properties of two interconnected components are matching. A list of such properties in the automotive domain can also be found in section 2.3.

The validation of static compatibility in the IIS is implemented with use of a combination of the Object Constraint Language (OCL)[2] Implementation and the EMF Validation Framework[3]. The constraints are added as annotations in the ecore meta model. Through the EMF code generator the required validation methods are generated and the validation can be executed in the resulting EMF editor.

[2] available at http://www.eclipse.org/modeling/mdt/?project=ocl

[3] see http://www.eclipse.org/modeling/emf/?project=validation for more information

Example Consider two components *C1* and *C2* which are connected through two ports each. The port o_{C1} is connected to i_{C2} and the port o_{C2} is connected to i_{C1}. Both components have a set of Ecore annotations, in which the compatibility constraints are defined with OCL. Here, the component *C1* receives data on port i_{C1} from *C2* and its output port o_{C2}. The annotation checks for every port of the component, whether the update frequency of the output port matches the update frequency of the receiving port. The annotation for component *C2* works similar but it checks the compatibility of the data type attribute of the ports.

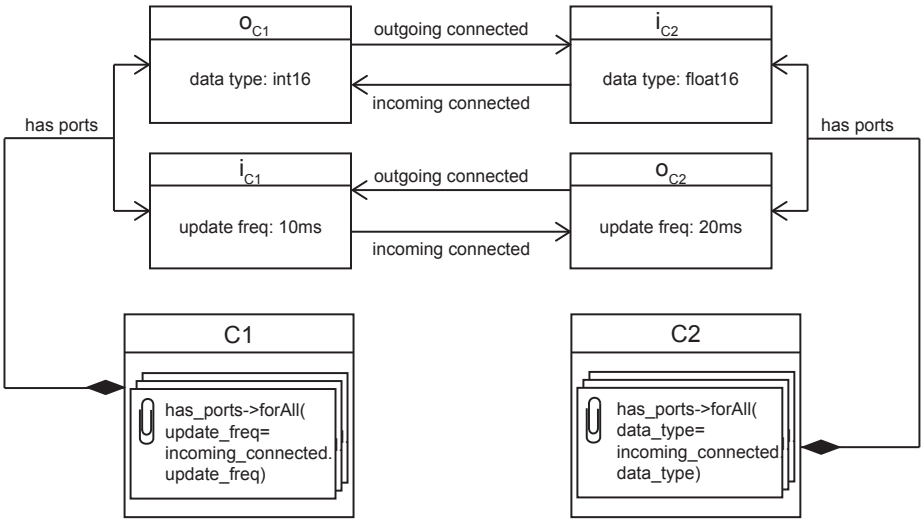


Figure 5.6: Two interconnected components with static compatibility constraints.

5.7.2 Behavioral Compatibility

The verification of dynamic compatibility in the design phase of software development is a complex task. The complexity of this task lies primarily in the requirements towards embedded software systems in the automotive domain like high cost pressure, restrictive safety requirements and real-time requirements, a high grade of componentization, software reuse and also the integration of legacy third party and customer components. Secondly, the verification takes place at a development stage in which parts of the components are not thoroughly defined. The dynamic compatibility checker (DCC) is a separate system, which is connected to the IIS through the compatibility checking adapter. The implementation of the dynamic compatibility checker with use of interface automata is described in detail in chapter 6.

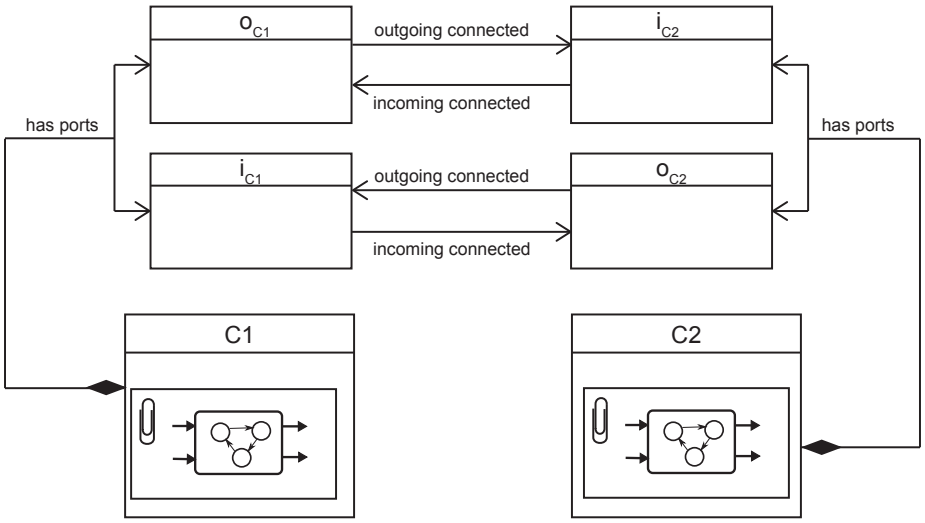


Figure 5.7: Two interconnected components with annotated dynamic behavioral models.

This section describes, how the DCC is connected to the IIS. Figure 5.8 shows the connection between the DCC and the adapter in the IIS. After a job is triggered through the IIS (*1* in fig. 5.8) the adapter generates an input file according to the components descriptions (*2* in fig. 5.8). The compatibility checker is the started (*3* in fig. 5.8) and reads the input files (*4* in fig. 5.8). After the compatibility check is completed, the results are read by the adapter (*5* in fig. 5.8) and presented to the user (*6* in fig. 5.8).

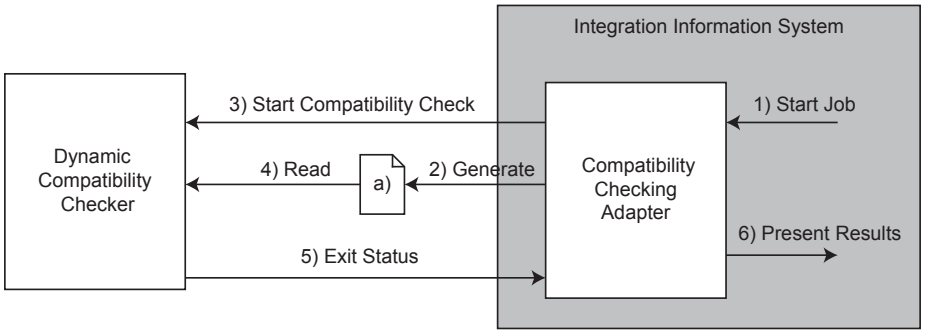


Figure 5.8: Dynamic compatibility checker and the according adapter in the IIS. The numbers denote the sequence of a compatibility verification job. File *a)* is the input file for the DCC.

The input files for the DCC are generated from the dynamic behavior descriptions, which are assigned to every component in the integration

model.

5.8 Integration Planning System

Similar to the compatibility checker form section 5.7.2, the integration planning system (IPS) is a separate system that is connected to the IIS through a dedicated adapter. Following the definition of the compound decision support system in Burstein and Holsapple, 2008, the IPS represents the solver element in the solver-oriented part of the DSS. The purpose of the IPS is to produce an integration sequence proposal based on the integration model. The theoretic background of the IPS with use of a game-theoretic solver is described in chapter 7. The implementation details such as communication phases and interchange file syntax of the interface between the IIS and game-solving IPS are described below.

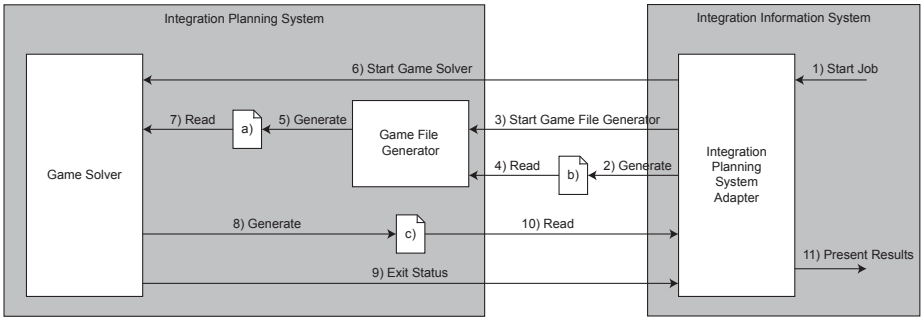


Figure 5.9: Integration Planning System with Adapter. Numbers denote the sequence of execution of an integration planning job. File *a)* is the game file, *b)* is the intermediate file for the game file generation and *c)* is the result file that is provided by the solver.

An IPS job is executed as follows: The user triggers the job through the IIS UI (*1* in fig. 5.9). The adapter generates an intermediate file (*2* in fig. 5.9), which is used by the game generator to produce a game file (*3,4 and 5* in fig. 5.9). When the game file is generated, the solver process is started (*6* in fig. 5.9), which reads the game file (*7* in fig. 5.9). The solver produces a solution file (*8* in fig. 5.9) and notifies the adapter on completion of the solving process (*9* in fig. 5.9). The solution file is parsed by the IPS adapter (*10* in fig. 5.9) and triggers the generation of the corresponding elements in the integration model (A new integration sequence is created with a set of integration step. The components are assigned to the integration steps according to the solution of the solver). The newly created integration sequence is then visible in the UI (*11* in fig. 5.9) and can be used in all other IIS modules.

There are three types of files involved in the IPS process, the intermediate game file, the game file and the solution file.

Intermediate File The intermediate file is used as input for the game generator. It is generated from the integration model and contains the following information:

1. Number of Components p.
2. Number of Possible Integration Steps s.
3. Component Dependencies d. This is a row of integers for each depending component p. The depending component takes the first place in the sequence.
4. Component Timeframes t. The availability timeframe for each component p can be defined in one or more rows. The first place of the sequence denotes the component. The possible integration steps can be selected in a single-value fashion or through intervals.
5. Resources r.
6. Resource Timeframes rt. The availability timeframes of the resources.
7. Component Resource Dependencies dr. A row of integers similar to the component dependencies d. Each row defines the dependencies between a component and a list of resources.

Game File The syntax of the game file is defined by the game solver, in this case the AGGSolver by Jiang, 2011. The game file generator produces a game file in the AGGSolver syntax from the intermediate file. This game file defines the final properties of the game: the players, their potential actions and the payoffs under each combination of the players' actions. The syntax of a game file for the AGGSolver is described below (cf. Jiang, 2011) and an example game file can be found in appendix B.1.

1. The number of players, n.
2. The number of action nodes $|S|$.
3. The number of function nodes, $|P|$.
4. Size of action set for each Player.
   This is a row of N integers: $|S_1||S_2|\ldots|S_N|$
5. Each player's action set. We have N rows; row i has $|S_i|$ integers which are indices of action nodes. Actions are indexed from 0 to $S-1$.

6. The Action Graph. We have $S + P$ nodes, indexed from $0 to S + P - 1$. The projected nodes are indexed after the Action nodes. The graph is represented as $(S + P)$ neighbor lists, one list per row. Rows 0 to $S-1$ are for action nodes; rows S to $S+P-1$ are for function nodes. In each row, the first number $|\nu|$ specifies the number of neighbors of the node. Then follows $|\nu|$ numbers, corresponding to the indices of the neighbors.

   We require that each function node has at least one neighbor, and the neighbors of function nodes are action nodes. The action graph restricted to the function nodes has to be a directed acyclic graph (DAG).

7. Types of functions. This is a row of $|P|$ integers, each specifying the type of mapping f_p that maps from the configuration of the node p's neighbors to an integer for p's "action count". The following types of mapping are implemented:

   - Type 0: Sum. i.e. The action count of a projected node p is the sum of the action counts of p's neighbors.
   - Type 1: Existence: Boolean for whether the sum of the counts of neighbors are positive.
   - Type 2: The index of the neighbor with the highest index that has non-zero counts, or $|S| + |P|$ if none applies.
   - Type 3: The index of the neighbor with the lowest index that has non-zero counts, or $|S| + |P|$ if none applies.

8. The payoff function for each action node. So we have $|S|$ subblocks of numbers. Payoff function for action s is a mapping from configurations to real numbers. Configurations are represented as a tuple of integers; the size of the tuple is the size of the neighborhood of s. Each configuration specifies the action counts for the neighbors of s, in the same order as the neighbor list of s.

   The first number of each subblock specifies the type of the payoff function. There are multiple ways of representing payoff functions; we (or other people) can extend the file format by defining new types of payoff functions. We define two basic types:

   - Type 0: The complete representation. The set of possible configurations can be derived from the action graph. This set of configurations can also be sorted in lexicographical order. So we can just specify the payoffs without explicitly giving the configurations. So we just need to give one row of real numbers, which correspond to payoffs for the ordered set of configurations.

If action s is in multiple players' action sets (say players i, j), then it is possible that the set of possible configurations given s_i is different from the set of possible configurations given s_j. In such cases, we need to specify payoffs for the union of the sets of configurations (sorted in lexicographical order).

- Type 1: The mapping representation, in which we specify the configurations and the corresponding payoffs. For the payoff function of action s, first give $Delta_s$, the number of elements in the mapping. Then follows $Delta_s$ rows. In each row, first specify the configuration, which is a tuple of integers, enclosed by a pair of brackets "[" and "]", then the payoff.

  In the example game file in appendix B.2, there are four players that correspond to the four components of the described system. Each player's action set has eleven action node. This means the game is used to determine an integration schedule with eleven possible integration steps.

Solution File The solution file is generated by the solver and displays the Nash equilibrium of the game. Listing B.3 shows an example solution file. The central part of the solution file is a list of row vectors that represent the equilibria of the game. A game with three players for example, each of which has two actions, and that the strategy profile in which Player 1 chooses his first action, Player 2 and 3 choose their second action, is an equilibrium. Then the output solution file will be:

```
1 0 0 1 0 1
```

The output varies slightly, depending on the algorithm that was used for solving the game, i.e. the global newton method or the simplicial subdivision implementations which are supported in the AGGSolver (cf. govindan2003global and van Der Laan, Talman, and Van der Heyden, 1987).

5.9 Visualization

Projects with a time span of several years and products with more the 100 components with up to 1000 atomic software functionalities are common in the automotive domain. The integration sequences for such products can have a considerable length, several interim releases and have a high degree of complexity.

The aforesaid complexity of these integration sequences make it difficult to evaluate their general feasibility and quality. It also complicates

an easy identification of problematic integration step, i.e., integration steps that cause significant effort.

An integration sequence visualization is an adequate solution to this problem. It enables integrators to identify potential risks in an integration sequence at a glance and without a deep knowledge of the system details.

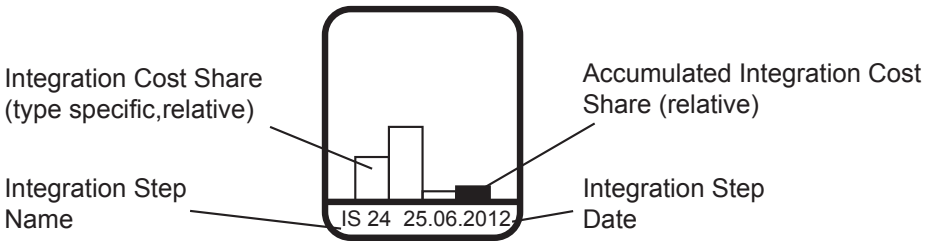


Figure 5.10: Integration step visualization element.

The integration visualization of the IIS is based on the sequential nature of the visualization subject. It consist of a series of containers, which represent the integration steps. The integration cost for each integration step is illustrated as bar chart inside of these containers. Figure 5.10 shows an example of an integration step. It is divided in two parts. The lower part contains a name field and a date field the upper part consists. The upper part is the main area of interest for the viewer. It displays the share of the integration cost of the particular integration step as a series of bar charts relative to the integration cost of the integration sequence. There are separate bar charts for different integration cost types (see section 7.2 for a description of these costs and the underlying measurement model) and a accumulated cost bar for the integration step.

These containers are aligned on the x-axis of the diagram according to their date inside of an array of possible integration time slots. The components availability time frames and the resource availability time frames are located below the integration steps and are visible on demand. Figure 5.11 shows and exemplary visualization of an integration sequence with five integration steps distributed over six possible integration time slots.

The integration steps can be highlighted to display additional information. The list of components for the selected integration step is displayed next to the integration sequence. This list can also be used to assign components to the integration step. If the component availability and resource availability time frames are enabled, they are also highlighted accordingly. If the integration step contains components with unsatisfied dependencies (when stubs are needed for integration), an arrow is directed to the integration step in which the necessary component is inte-

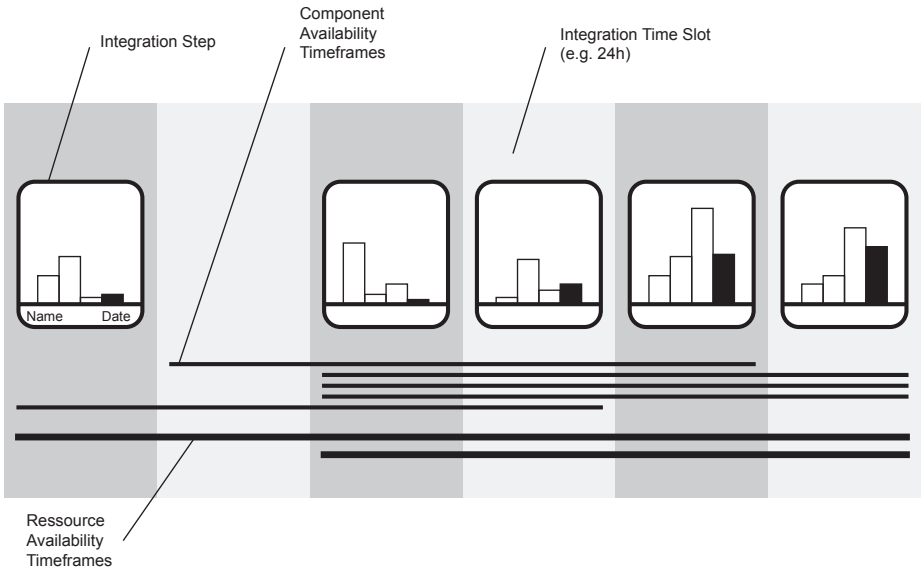


Figure 5.11: Standard integration sequence visualization.

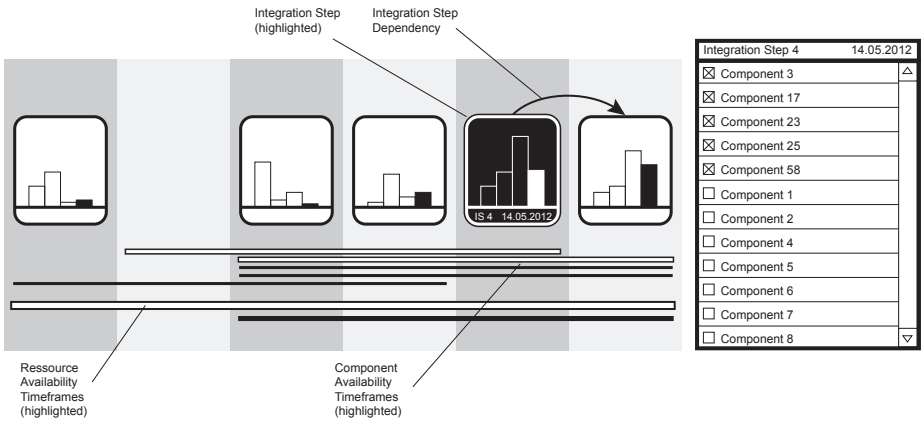


Figure 5.12: Integration sequence visualization with integration step selection.

grated. In figure 5.12, integration step *IS 4* contains a component, which depends on another component that is integrated in the following integration step.

5.10 Build System

The integration planning system with its automatic determination of feasible integration schedules from section 5.8 showed that complex tasks in the context of software integration can be made significantly more user friendly. Building the software system is another process during the software integration, which can benefit from tool support. According to Schäuffele and Zurawka, 2010, a build of an automotive software system includes

- generation of program and data in an executable format for the micro processor
- generation of the software documentation
- generation of description data for production and service tools, like diagnosis, software parameterization or flash programming tools.

Currently the build requires a lot of repetitive manual action. The integrator selects the required components for the active integration step from the source repository and copies them into a designated integration space. He then generates the three parts of the build as described above. Due to the number of software components and the complexity of software system the compilation and linking process can take several hours. This procedure has to be repeated if errors are found in the build, i.e. the build did not complete successfully. This can be due to software errors but also when the build was constructed with errors, e.g. with wrong or missing components.

The mostly manual part of selecting the content of the build for generation can be supported efficiently through an implementation of an automatic build process. During this automated build process the information, which is required to build a specific set of components is stored in the IIS. This information can later be used to perform a build automatically. When all components that need to be integrated in a particular integration step are available for integration (i.e. they have the status *ready to integrate* in integration management) and the required build information is present all prerequisites are fulfilled to perform the build without any attendance of the integrator.

Chapter 6

Verification of Dynamic Compatibility

A central part of the integration information system is verification of component compatibility in the design phase (cf. section 3.2.1 and section 5.7). The characteristics of software development in the automotive industry, which are described below, have a heavy impact on the integration phase of the software product.

Software development in the automotive industry can be characterized as follows.

- Development is under high cost pressure.
- The number of components is constantly rising.
- Component variations increase the variability of component behaviors.
- Legacy, customer and third-party components need to be integrated.
- The software has to satisfy real-time requirements.
- The software has to fulfill strict safety requirements.

Safety and real-time requirements have to be fulfilled by the system as a functioning whole. Therefore it is vital that any faults in the interactions of the components need to be avoided. These faults are discovered in the integration phase of the software development cycle.

The rising number of components, the variability in the component behaviors and the need to integrate legacy, customer or third-party components with not fully known properties makes integration of automotive software system a complex task.

The high cost pressure for developments in the automotive domain does not allow the redesign of the system architecture in case of the discovery of incompatibilities between components. Every delay in the development process can lead to further delays in the product development or significant contractual penalties.

The virtual integration methodology proposes a compatibility checking process that aims to reduce the risk of unaccounted, additional costs for error removal in the integration phase. This is done through a design verification method that is focused on the compatibility between the components in the automotive software system.

The following core requirements of such a design verification formalism were defined together with the research cooperation partners from Continental Automotive GmbH.

- It shall be able to describe all factors, which influence component compatibility, especially considering real-time requirements.
- Verification of compatibility between components shall be completed within reasonable time.
- The formalism shall be applicable independent from system architecture.
- A seamless integration in existing development processes shall be possible.
- The formalism shall be comprehensible and easy to use.

The following presents a feasibility study on the use of interface automata for the foresaid task. Interface automata are a light-weight formalism that capture interface behavior and enable a verification of interface compatibility within a reasonable timeframe. They further offer a type system for component interaction. The verification of compatibility between interface automata as well as the conformity verification of component interaction types is based on game-theoretic foundations.

6.1 Model

This section gives an introduction of the interface automata and timed interface automata definitions and how the behavioral verification in the integration can be achieved with use of these formalism.

6.1.1 Interface Automata

Interface Automata (IA) (cf. de Alfaro and Henzinger, 2001a) capture external behavior of components in component-based systems and the

compatibility between components can be verified in linear time. An interface automaton $P = \langle V_P, V_P^{init}, A_P^I, A_P^O, A_P^H, T_P \rangle$ consists of the following elements:

- V_P is a set of states
- $V_P^{init} \subseteq V_P$ is a set of initial states. We require that V_P^{init} contains at most one state. If $V_P^{init} = \emptyset$, then P is called empty.
- A_P^I, A_P^O, and A_P^H are mutually disjoint sets of input, output, and internal actions. We denote by $A_P = A_P^I \cup A_P^O \cup A_P^H$ the set of all actions.
- $T_P \subseteq V_P \times A_P \times V_P$ is a set of steps.

Figure 6.1 shows an example interface automaton.

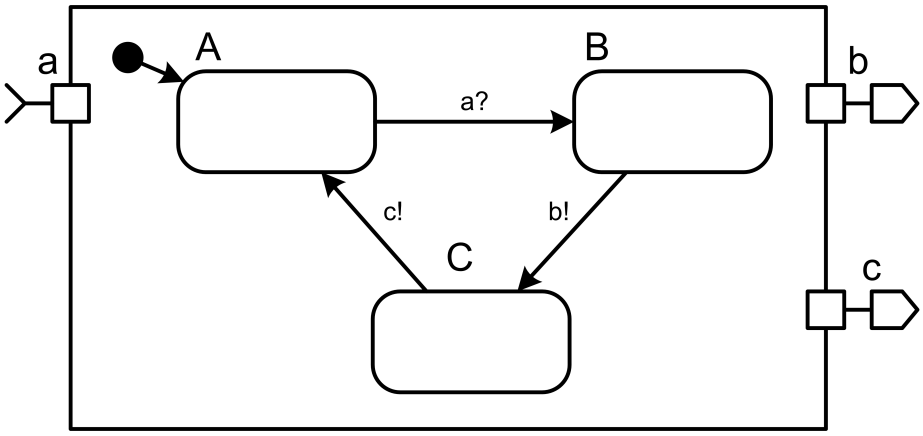


Figure 6.1: Example of an interface automaton. The automaton consists of the states *A*, *B* and *C*. *A* is the initial state of the automaton. The automaton has an input port *a* and two output ports *b* and *c*. The input action *a?* is taken between the states *A* and *B*. The output actions *b!* and *c!* are executed between state *B* and *C* and between *C* and *A* respectively.

6.1.2 Timed Interface Automata

Verification of real-time components requires a consideration of dynamic behavior in the formal model. Interface automata can be extended to timed interface automata to model and to verify temporal aspects of components. Figure 6.2 shows an example interface automaton.

A timed interface automaton (TIA) is a tuple

$$A = \left[Q_A, q_A^{init}, X_A, Acts_A^I, Acts_A^O, Inv_A^I, Inv_A^O, \rho_A\right]$$

consisting of the following components de Alfaro, Henzinger, and Stoelinga, 2002.

- Q_A is a finite set of locations.
- $q_A^{init} \in Q_A$ is the initial location.
- X_A is a finite set of clocks.
- $Acts_A^I$ and $Acts_A^O$ are finite and disjoint sets of input and output actions, respectively. Let $Acts_A = Acts_A^I \cup Acts_A^O$ denote the set of all actions of A.
- $Inv_A^I : Q_A \mapsto \Xi(X_A)$ maps each location of A to its input invariant.
- $Inv_A^O : Q_A \mapsto \Xi(X_A)$ maps each location of A to its output invariant.
- $\rho_A \subseteq Q_A \times \Xi[X_A] \times Acts_A \times 2^{X_A} \times Q_A$ is the transition relation. For $(q, g, a, r, q') \in \rho_A$, the locations q and q' are the source and destination of the transition, $g \in \Xi[X_A]$ is a guard on the clock values that specifies when the transition can be taken, $a \in Acts_A$ is an action labeling the transition , and $r \subseteq X_A$ is a set of clocks that are reset by the transition. We require the transition relation to be deterministic: for all $q \in Q_A$ and $a \in Acts_A$, there is at most one tuple of the form (q, g, a, r, q') with $(q, g, a, r, q') \in \rho_A$.

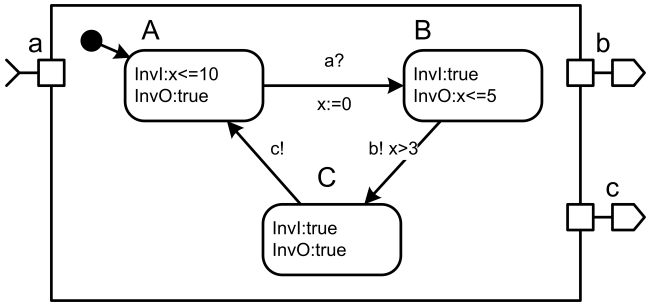


Figure 6.2: Example of a timed interface automaton. It partly resembles the example from figure 6.1. Additionally, it contains a clock variable *x* and the states have invariants for input and output attached to them. For state *A*, this means that input action *a?* can only be taken, if the clock *x* is 10 or less. The clock is reset to 0, when the input action *a?* is executed. For state *B*, the output *b?* can only be taken, if the clock *x* is five or less. No invariants are defined for state *C*, so the action *c!* can be taken at any time.

6.1.3 Modeling Integration with Interface Automata

The description of component behavior with use of interface automata makes a Virtual Integration with regards to behavioral compatibility pos-

sible. Therefore, the integration model features annotations for the interface automata representation of every component. Virtual Integration is carried out by successively building the composition of single components with a system, which was previously composed from other components.

The advantage of interface automata, compared with other formalisms, is that building the composition automaton from two automata is equivalent to the verification of compatibility between these two automata. Additionally, interface automata have a natural way of describing interface behavior. The so-called *optimistic* approach to composition implies that in order to form a compatible composition from two interface automata, they have to work together in at least *one* environment. This results in a reduced complexity and size of the automata, compared to the *pessimistic* approach, where they have to be compatible in *any* environment.

The process of building a composite automaton includes several steps. First, the two interface automata have to be *composable*, which means that their action signatures have to match. Alfaro and Henzinger define the composition of two interface automata only if their actions are disjoint except that an input action of one may coincide with an output action of the other. The two automata will synchronize on such shared actions and asynchronously interleave all other actions. Two interface automata P and Q are composable if de Alfaro and Henzinger, 2001a:

$$A_P^H \cap A_Q = \emptyset \quad A_P^I \cap A_Q^I = \emptyset$$
$$A_P^O \cap A_Q^O = \emptyset \quad A_Q^H \cap A_P = \emptyset.$$

We let shared$(P, Q) = A_P \cap A_Q$.

The resulting product automaton $P \otimes Q$ is prerequisite for the composition of two interface automata. This product automaton contains so-called *error states,* in which an input assumption of one component is violated by the output of the other component. The components can still be compatible if the environment can prevent the product automaton to enter such a state. Since the environment can only influence input actions of the product automaton, some states cannot be avoided. These states are called *incompatible states* and are removed from the product automaton to form the composite automaton $P \parallel Q$.

Following de Alfaro and Henzinger, 2001a, two interface automata P and Q are compatible iff (a) they are composable and (b) their composition is not empty.

For this paper, we define two operators, which denote the compatibility between components:

Def. 1. *$X \odot Y$ denotes compatibility between X and Y.*

Def. 2. *$X \oslash Y$ denotes incompatibility between X and Y.*

In general, compatibility between two interface automata is defined as:
$P \odot Q, if\ P \parallel Q \neq \emptyset$ and $P \oslash Q, if\ P \parallel Q = \emptyset$

6.2 Verification Process

Compatibility between two interface automata or timed interface automata is verified by solving (timed) games between the composition automaton and the environment. In de Alfaro and Stoelinga, 2004, the players *Output* and *Input* are defined. *Output* represents the composition, *Input* represents the environment. When two interfaces are composed, the composition can contain error states, which occur if one component produces output, which violates input assumptions of the other component. Two interfaces are compatible if the input player, who chooses the inputs of the composition, has a strategy to avoid all error states.

6.2.1 Example

This section contains an example of the verification of dynamic compatibility between two components. The behavior of the two components is modeled with interface automata. These automata are illustrated in figure 6.3. Interface automaton 1 (IA_1) has two input ports, with accept the inputs A and B, and two output ports, which provide the outputs X and Y. The automaton will produce Y after it receives input B from its initial state. If it receives input A it will produce X. A second instance of A will produce an additional X whereas receiving B will trigger the output Y.

Interface automaton 2 IA_2 has four input ports, which accept the inputs A, B, X and Y. From its initial state, the automaton can accept input A or input B. In the subsequent state only the input X is accepted. After it received X it will again accept inputs A or B. In the following state (state 4) it will accept input Y.

Inputs A and B are provided by the surrounding environment of the two automata, inputs X and Y for IA_2 are provided by IA_1.

Figure 6.4 shows the first error states of interface automaton 1 and 2. IA_1 is in the state 3 after receiving input B and triggers the output Y on the transition to state 4. IA_2 is in state 2, since it accepts either A or B to move from state 1 to state 2. Being in state 2, IA_2 will now only accept input X to move to state 3. Since IA 1 produces Y this means that state 3 of IA_1 is a so-called error state.

In figure 6.5, the second error state is illustrated. The two automata received the input A at their initial state. After that IA_1 provided output

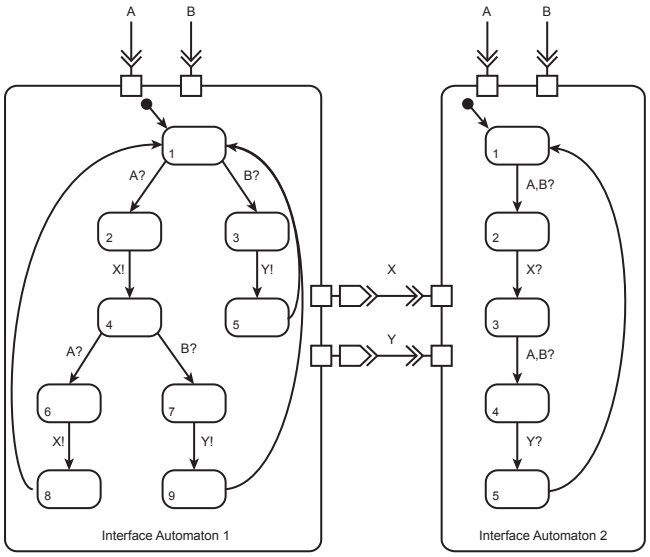


Figure 6.3: Interface Automata 1 and 2 before verification.

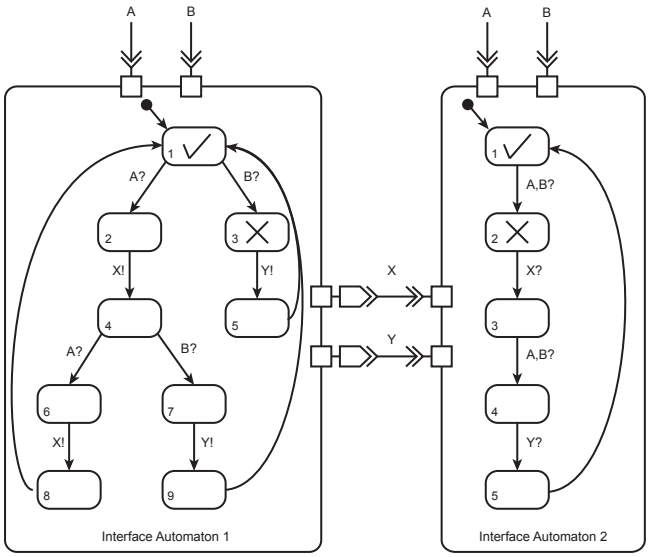


Figure 6.4: The first error state (state 3 in IA_1 and state 2 in IA_2) in the verification of behavioral compatibility between interface automaton 1 and 2.

X, which is conform to the input assumptions of IA_2. A second instance of input A lets IA_1 move into state 6 and IA_2 into state 4. Interface automaton 1 produces X from state 6 and interface automaton 2 will only accept Y from state 4.

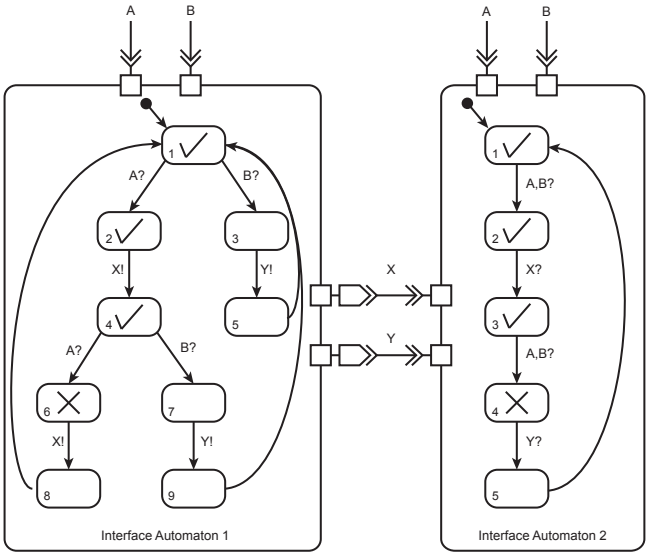


Figure 6.5: The second error state (state 6 in IA_1 and state 4 in IA_2) in the verification of behavioral compatibility between interface automaton 1 and 2.

The definition of compatibility that Alfaro et. al. defined in de Alfaro and Henzinger, 2001a states that two interface automata are compatible if there exists an input strategy for the environment so that the two automata can work together. Figure 6.5 shows that in our example, the environment can provide the inputs in a way so that IA_1 and IA_2 are working together. In that case, IA_1 provides the inputs X and Y for IA_2 according to IA_1's input assumptions. The interface automata are therefore compatible.

If two interface automata are compatible, it is possible to describe them as a composed interface automaton. This composed automaton in shown in figure 6.7.

6.2.2 Verification Process Example

Figure 6.8 shows the verification of behavioral compatibility and integration of five components $C_1 \ldots C_5$ to a composition C_{System}. Therefore, every component has to be described with an interface automaton. The integration sequence consists of four sequential integration steps. There

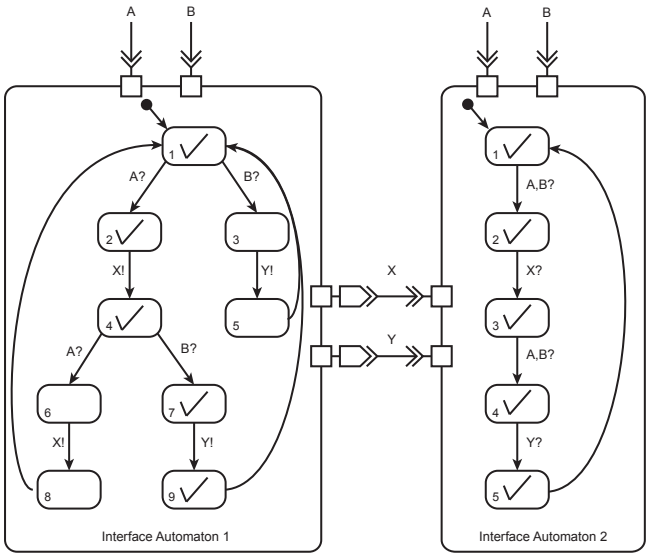


Figure 6.6: A successful input configuration for IA_1 and IA_2.

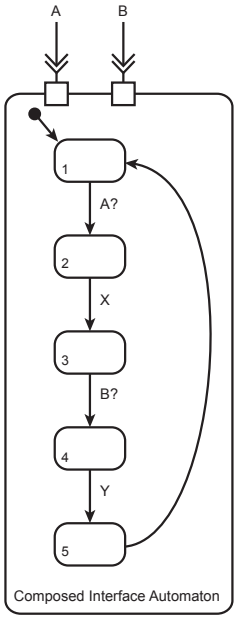
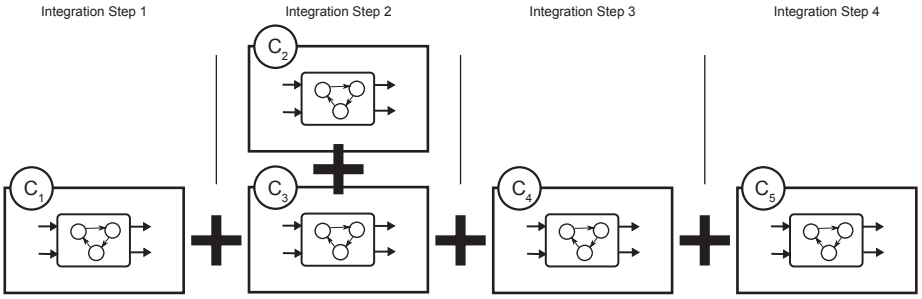
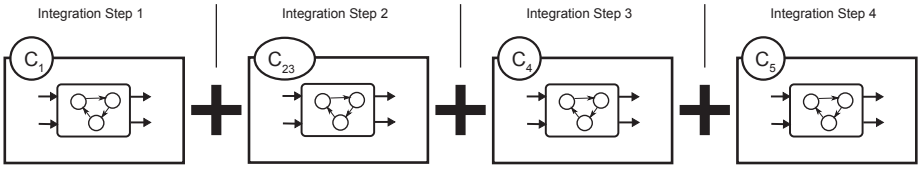


Figure 6.7: The composed interface automaton for IA_1 and IA_2.

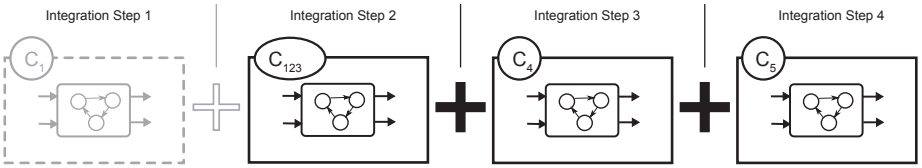
are two components to be integrated in the second integration step, all other steps contain only one component. The target of the verification process is the composition of the interface automata of all components.



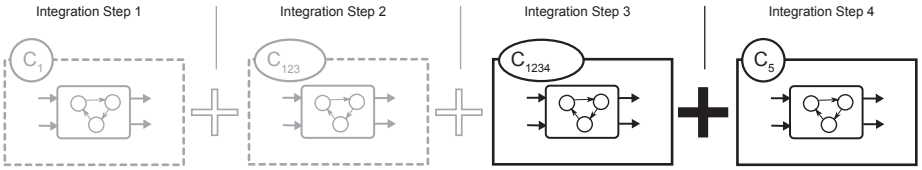
(a) Before Verification



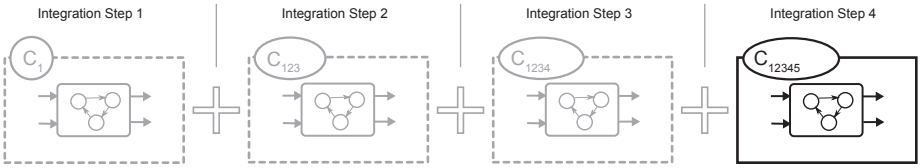
(b) Integration of C_2 and C_3 to C_{23}, $C_2 \odot C_3, if \exists\, C_2 \| C_3 \neq \emptyset$



(c) Integration of C_1 and C_{23} to C_{123}, $C_1 \odot C_{23}, if \exists\, C_1 \| C_{23} \neq \emptyset$



(d) Integration of C_4 and C_{123} to C_{1234}, $C_4 \odot C_{123}, if \exists\, C_4 \| C_{123} \neq \emptyset$



(e) Integration of C_5 and C_{1234} to $C_{12345} = C_{System}$, $C_5 \odot C_{1234}, if \exists\, C_5 \| C_{1234} \neq \emptyset$

Figure 6.8: This figure illustrates the subsequent integration of components and the verification process through building the automata compositions. At integration step 2, the components C_2 and C_3 have to be integrated before they can be integrated with C_1.

6.3 Interface Automata Case Study

The applicability of interface automata in the Virtual Integration methodology was examined in a case study. The example is an anonymized version of a project from the automotive industry supplied by iNTENCE automotive electronics. It is divided in its basic components and the components' interface behavior was modeled with interface automata. The components were virtually integrated to verify their compatibility.

6.3.1 Purpose

The module is used for diagnostic purposes during vehicle test. It logs sensor data on an usb storage device for long-term inspection of internal system behavior. The communication to the sensor controller is implemented over LIN (Local Interconnect Network) bus. The module itself is controlled over buttons on a handheld device. It also features a screen, which displays the current state of operation.

6.3.2 Modeling

The components were identified based on the system architecture description. It consist of the following components.

- OS: A minimal operating system initializes components and schedules the tasks.
- Timer: Generates timestamps for log values and triggers software timers.
- LIN Communication: Retrieves LIN messages.
- Controller: Controls logging, LIN reception and user interface.
- I/O: Controls input and output operations.
- Logging: Generates log messages from LIN frames, display output.
- USB Driver: Writes log messages to USB device.

The interface descriptions for the components provide information about the communication structure of the components. They define the input and output ports of the components. The system architecture description provides information on the interconnections of the individual components. Together with the functional models of the components it was possible to define the behavior of the component interfaces.

Interface Behavior Models

Adler et al. present a tool to check and compose interface automata in Adler et al., 2006. The *Tool for Interface Compatibility and Composition* (TICC) is used to describe the software components using the interface automata formalism. Figures 6.10 to 6.14 show the automata of a subsystem, including the *LIN*, *Logging* and *Timer* modules. The automata were successfully composed using TICC. The composite automaton $C_{Subsystem}$ is composed from

$$C_{Subsystem} = C_1 \| \{C_2 \parallel [C_3 \parallel (C_4 \parallel C_5)]\}.$$

The subsystem with the interconnections between the components is shown in figure 6.9. Figures 6.10 to 6.14 show the interface automata of the subsystem's components.

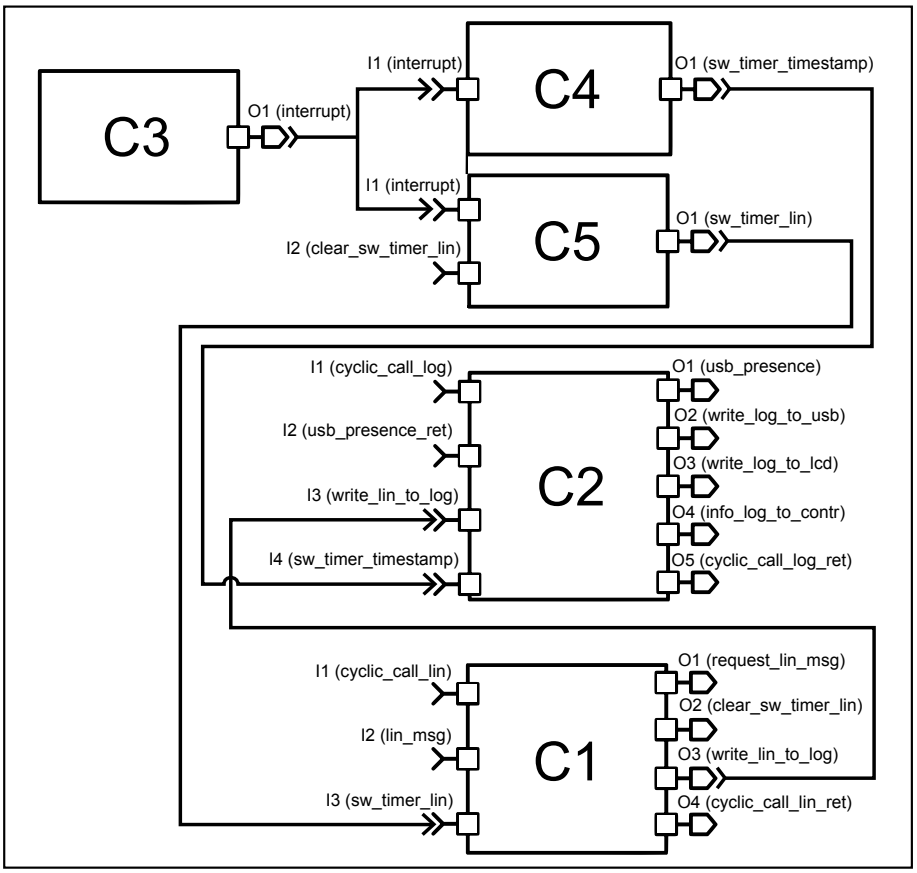


Figure 6.9: The components of the analyzed subsystem and their interconnections. Figures 6.10 show the graphical representations of the corresponding interface automata.

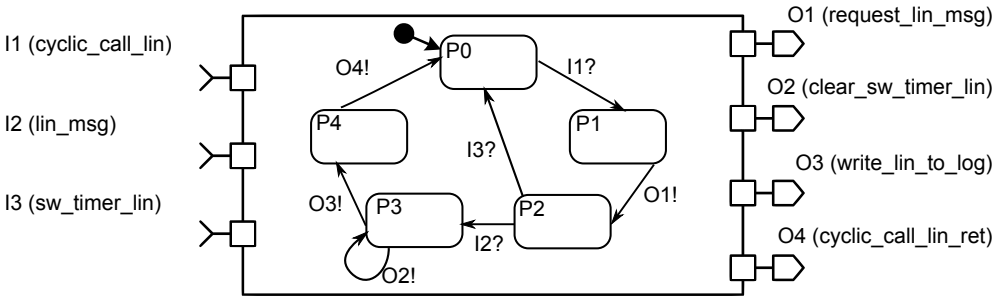


Figure 6.10: C1 (LIN driver automata): The LIN driver manages the LIN communication of the module.

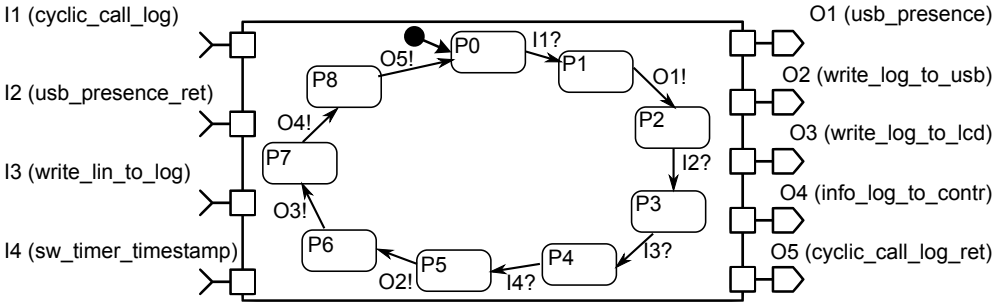


Figure 6.11: C2 (Logger automata): The logger is the main routine of the system. It checks the usb status, polls the data from the LIN driver, generates log messages with timestamps and sends the log message to the usb driver and to the lcd driver.

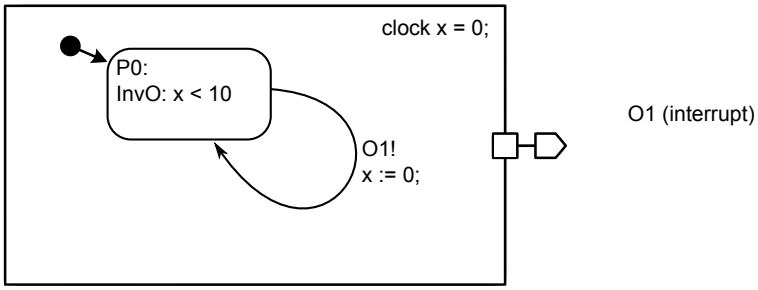


Figure 6.12: C3 (Hardware timer automata): The hardware timer sends an interrupt to the software timers with a period of ten milliseconds.

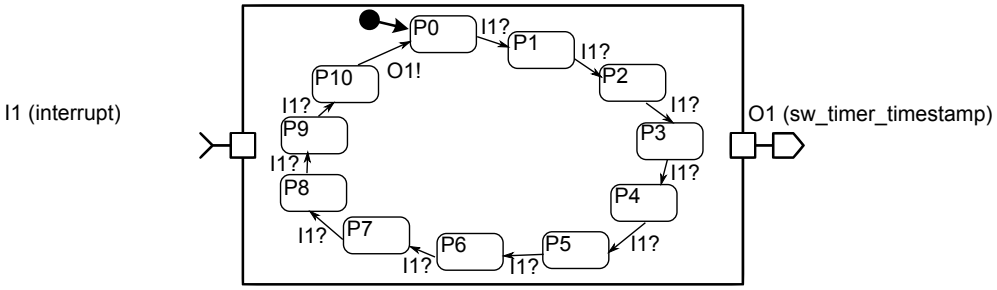


Figure 6.13: C4 (Timestamp software timer automata): The software timer for the logging timestamps generates a new timestamp every 100 milliseconds.

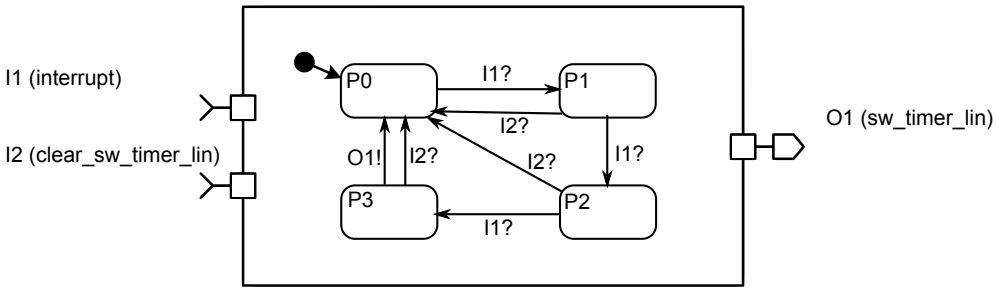


Figure 6.14: C5 (LIN software timer automata): The LIN software timer generates a message for the LIN driver, if the timer exceeds 30 milliseconds.

Chapter 7

Integration Games

THE integration phase in the development of component-based embedded systems in the automotive industry is characterized mainly by its time pressure and the its dependence on individual knowledge of the system. The integration strategy determines at which point in time the components of the component-based system are integrated. It also determines, which resources, e.g. human resources or tools, shall be used to accomplish the integration. Generally speaking, the quality of the integration strategy is based on the individual expertise of decision makers, like project managers or integrators. The quality of this strategy is also influencing the risk that bug fixing measures are required and the extend, which is required to implement them.

The integration strategy has also an effect on the cost of the integration procedure because it determines factors like the overall number of stubs and the length of the integration phase.

Due to the nature of a component-based system - the distribution of complex functionalities over several components - the interconnections of the components play a very important role for the functionality of the overall system. Integrators have to take these interconnection into account during the assembly of these systems, they have to satisfy the components' dependencies. This can be done either by arranging the components in an order which conforms to their dependencies or by introducing temporary stubs which simulate the required components. The first approach is difficult to accomplish due to dependency cycles, which are common in automotive applications. The introduction of stubs on the other hand will increase development and test effort for the software product. Besides the dependencies between components there are restrictions, which are connected to the project environment of the product like the maximum length of the integration sequence and human resource allocation during the integration phase. Also, there are considerations regarding the testability of the system during integration test.

General concepts for integration like top down integration or bottom up integration (see section 2.5 for in-depth descriptions) exist but these methods aim to give integrators a more general direction for the integration and provide rather vague instructions for the integration phase. These methods fail when there is a need to determine an explicit integration sequence for a particular system. Other approaches provide means to determine integration sequences based on special attributes of such a sequence.

In Tai and Daniels, 2002, Briand, Labiche, and Wang, 2003 and Le Traon, Jéron, Jézéquel, and Morel, 2002, methods for integration scheduling are presented that focus on stub minimization. This means, the focus during the definition of the integration schedule is to keep the effort for stub development and test as low as possible. As a consequence, these approaches try establish an integration order that is conform to the dependency direction between the components.

This type of approach is extended through test case mapping in Borner and Paech, 2009. Functional test cases that were designed in the system design and implementation phase are included in the decision process for the integration schedule. The integration order of the components is aligned to the test cases to improve the quality and cost effectiveness of the test phase.

Another aspect of integration schedule optimization is shown in Binder, 1999 and McGregor and Sykes, 2001. Their approaches aim to arrange the components integration according to use cases.

The following section will show that for a cost effective integration phase it is not sufficient to focus on the optimization of single aspects of the integration sequence but to incorporate a combination of influences on integration schedules in the integration planning optimization process.

7.1 Concept of Integration Games

This section describes the concept of integration games and the general outline of the approach. It consists of following parts:

- Model for integration
- Integration cost measurement
- Integration planning problem definition
- Solution concept for the integration planning problem
- Implementation of the optimization method through game theoretic structures

The following section presents a formalization of the integration phase and its elements. This means that the underlying structures of a system during integration are abstracted. This model includes information about architectural properties of the software like component dependencies, as well as project-specific parameters like release dates and human resources.

A set of cost metrics were designed in cooperation with industry partners, which make it possible to capture key factors of influence on integration costs. Examples are development of component stubs or violation of project deadlines.

With use of the integration model and the cost metrics it is possible to define the so-called integration planning problem. The integration planning problem belongs to the class of combinatorial optimization problem.

The final step is the implementation of a solution method for the integration planning problem. The concept of integration games is introduced, which makes a rapid solution of the integration planning problem possible.

7.2 Integration Cost Measurement

Through extensive interviews with experts (cf. Schorer, 2011a) of the research project's industry cooperation partners, it is possible to define the main factors of influence for an integration sequence as

- the length of the complete sequence,
- the number of components to be integrated in each integration step,
- the number of unsatisfied dependencies,
- the availability and delivery date of each component and the
- available human resources for integration.

According to these factors of influence it is possible to define a notion of cost for the integration. Each factor influences the overall cost for the integration. Before an integration sequence can be measured, an abstract model of the system and its environment is introduced on which this metric is applied. A profound formalization of the costs is necessary to design such a metric for integration sequences. With this metric an evaluation of integration sequences with regards to the integration costs is possible.

7.2.1 Model

A formal model of the target software system as well as project and integration relevant attributes provides the basis to calculate the integration

costs. The following terms are used throughout the formal definition of this model.

Definition 1. *Component*
The term component is used in this thesis according to the OMG's definition of a component Object Management Group, 2007. The document "(. . .) specifies a component as a modular unit with well-defined interfaces that is replaceable within its environment." The component is uniquely defined by the interfaces it provides to its environment and the interfaces it requires from its environment. The functionality of the component is not visible, the component is treated as a black box.

Definition 2. *Component Dependency*
Following definition 1, a dependency between components occurs between components when a required interface of one component is provided by another component. "Although it may be dependent on other elements in terms of interfaces that are required, a component is encapsulated and its dependencies are designed such that it can be treated as independently as possible."

Definition 3. *Integration step*
An integration step is an atomic unit of time which is used to model an sequential ordering of integration activities. The length in time of an integration step is not fixed. However, the length of one day per integration step is a practical and realistic measure. The integration phase consists of exactly the number of integration step that are needed to assign every atomic time interval to an integration step (e.g.: If the integration phase would consist of 14 days and the desired integration step size is one day, there would be 14 integration steps). Components are assigned to an integration step to reflect the sequence and temporal position in the integration phase.

Definition 4. *Component availability timeframe*
The component availability timeframe denotes a interval during the development process. The term availability means in this availability for integration. It starts after the components functionality is module tested. The component's release date marks the end of the availability timeframe.

Definition 5. *Resource*
A resource is a enabling factor for component integration during the integration phase. This can be a human resource, such as a specialist or responsible person, or a limited material resource like a integration testing framework. Each component requires at least one resource to be integrated.

Definition 6. *Resource availability timeframe*
The availability timeframe is very similar to the component availability

timeframe except the fact that there can be multiple availability intervals of a single resource.

Definition 7. *Integration sequence*
The integration sequence is the set of integration steps for the integration phase of a software product with all its components assigned to integration steps.

Definition 8. *Length of an integration sequence*
The length of an integration sequence is measured from the start of the integration phase (from the first integration step) until the last integration step that has any components assigned to it. Trailing integration steps are not counted since they do not require any integration activities.

Definition 9. *A System under Integration (SI) is derived from the term system under test, which denotes a software system that is currently undergoing test procedures. A system under integration is therefore a set of components that is being integrated into a functioning whole. A SI is a tuple $SI = (C, S, R, m_C, m_S, m_D, m_R, m_{RC}, m_A)$ where:*

- *C is the set of components. Each component $c \in C$ a component, and C the set of distinct components.*
- *S is the set of integration steps. Each integration step $s \in S$ an integration step, and S the set of distinct integration steps.*
- *R is the set of resources. Each resource $r \in R$ a resource, and R the set of distinct resources. A resource r is a tuple $r = (perc, rate)$, with the allocation percentage $perc$ of the resource towards the SI and the hourly rate in $\frac{\$}{h}$.*
- *$m_C : C \mapsto S$ maps all components C to the integration steps S.*
- *$m_S : S \mapsto I$, with $I = \{1, 2, \ldots, |S|\}$, maps all integration steps s bijectively to the ascending integers, which represent the temporal order of the integration steps.*
- *$m_D : C \mapsto C$ maps depending components to the components they depend on. This mapping is irreflexive and surjective.*
- *$m_R : R \mapsto S$ maps all resources to the integration steps S according to their availability.*
- *$m_{RC} : C \mapsto R$ maps components C to their required resources R.*
- *$m_A : C \mapsto S$ maps all components C to the integration steps S according to their availability timeframe.*

Figure 7.1 shows the concept of a system under integration. The main part is the set of integration steps, which is located in the center of the illustration. The set of components is arranged above the integration steps, the set of resources is below.

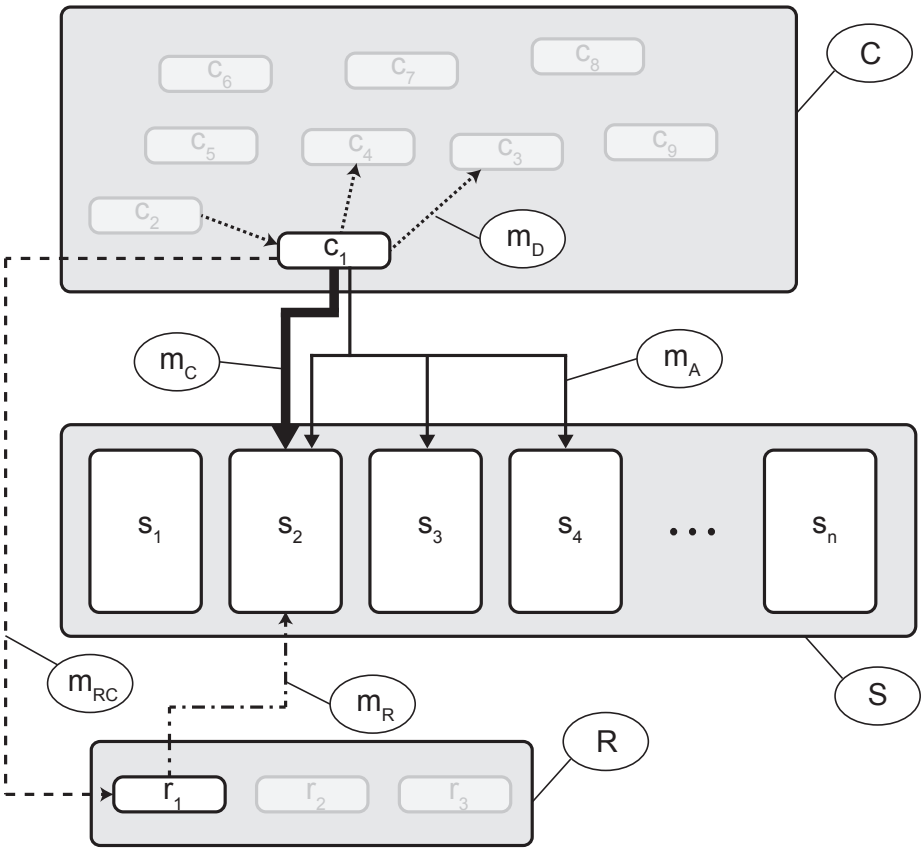


Figure 7.1: A system under integration and its components. See definition 9 for details.

7.2.2 Metrics

This section defines the metric which are used to evaluate an integration sequence in integration measurement. These metrics are also the foundation to the optimization of the integration planning process. They represent the different dimensions of the optimization problem.

Metrics are a widely used instrument in software development. They are used to measure software quantity, software complexity, software quality, requirement and design quality, code quality and complexity, software testing, software productivity and maintainability (cf. Baumgartner, Sneed, and Seidl, 2010). In Singh, Singh, and Singh, 2011 an

overview of commonly used metric is given.

For software integration measurement, the effort for a particular integration process is inspected. A metric for the execution of an integration plan belongs the software productivity metric category. In the following, the single aspects of the integration metric are presented.

Sequence Length

The overall cost of the integration phase is heavily influenced by its overall makespan. The compression of this timeframe results in an imminent cost reduction of this factor.

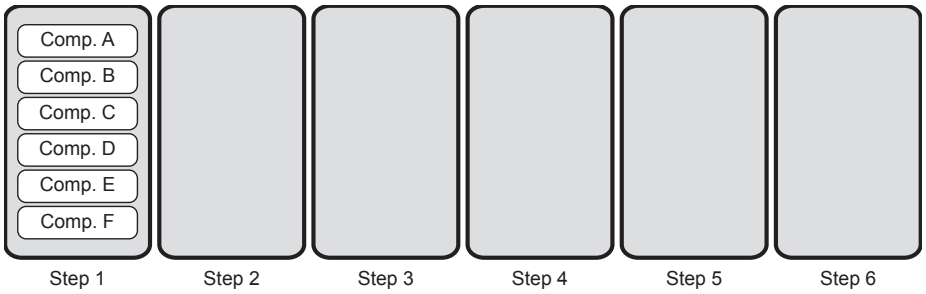


Figure 7.2: An optimal integration sequence concerning integration sequence length. The components are arranged in order to use the minimum number of integration steps.

The cost for the temporal length CT_L of a SI is

$$CT_L\left(SI\right) = \sum_{r \in R} \frac{\sum_{s \in \overline{S}} perc_r \, rate_r}{\sum_{s \in S} perc_r \, rate_r},$$

$$\text{with } \overline{S} = \{s_1, s_2, s_3, \ldots, s_x\},$$

$$s_x = \{s \in S | \forall i \in \{i+1, i+2, \ldots, |S|\} m_C(s_i) = \emptyset\}$$

with the set of used steps $\overline{S}$, the possible integration sequence and set of possible integration steps S. This cost function formulates the costs relative to the maximum possible costs (worst case).

Example: The integration phase of a SI has 100 possible steps, which means that the integration phase extends over a maximum of 100 days (given that one step has a length of one day). During the integration phase, there are two employees allocated to the project. The first employee, r_1, is working at 80% on the project at an hourly wage of 100$. The second employee, r_2, is a specialist who is allocated to 20% on the project at an hourly wage of 200$. Both employees are assigned in a consistent way over the integration phase.

Due to a conservative project planning and a high reuse factor of the software components, the last integration step could be completed at day 75 of the integration phase. Therefore, the absolute cost reduction would be:

$$\begin{aligned}\triangle CT_L &= 24\text{h} \times 25\text{d} \left(\frac{100\ \$}{\text{h}} \times 0.8 + \frac{200\ \$}{\text{h}} \times 0.2 \right) \\ &= 72,000\ \$\end{aligned}$$

The relation to the previously planned costs would be:

$$\begin{aligned}CT_L &= \frac{24\text{h} \times 75\text{d} \left(\frac{100\ \$}{\text{h}} \times 0.8 + \frac{200\ \$}{\text{h}} \times 0.2 \right)}{24\text{h} \times 100\text{d} \left(\frac{100\ \$}{\text{h}} \times 0.8 + \frac{200\ \$}{\text{h}} \times 0.2 \right)} \\ &= 0.75\end{aligned}$$

Potential Extensions: The hourly rates are rising during the project lifetime. Therefore, either the increasing human resource costs have to be anticipated at project planning or these increases have to be handled with during the integration phase.

A notion of supply and demand of human resources is incorporated. Finishing earlier will yield a higher cost reduction when there is a high demand for that type of resource and vice versa.

Testability

Increasing the average number of components per integration step is a direct consequence from shortening the integration sequence length. Furthermore it can occur due to other restrictions, like the availability of components. The testability[1] of the system under test decreases with the number of simultaneously integrated components and consequently the test effort increases to reach the required test coverage.

In Binder, 1994, Binder defines testability as “the relative ease and expense of revealing software faults”.

In the context on integration testing, the testability is strongly connected to the complexity of the components interfaces. In the SI model the components interfaces are represented by the dependency relation between m_D components. Therefore, the testability within a integration step is correlated to the number of dependencies between the components that shall be integrated in this step and the components that are already integrated.

Figure 7.3 shows an exemplary integration sequence of six components (Component A-E), which can be integrated in a maximum number

[1] Only the integration test is considered in this case. All components are unit tested before their integration.

of six integration steps (Step 1-6). In this example an optimal configuration with regards to the testability of the system is present, every integration step contains a single component.

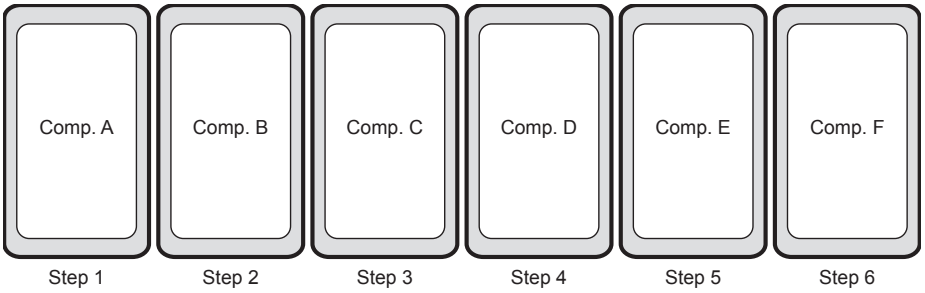


Figure 7.3: An optimal integration sequence concerning testability. The components are arranged in order to receive a minimum number of components per integration step.

The testability cost CT_T of the system under integration is defined as

$$CT_T(SI) = \sum_{s \in S} \frac{|m_C(s)|^2}{|C|^2}$$

with the number of components $|m_C(s)|$ in step s and the number of components $|C|$ of the complete integration sequence.

Example: Given an example system under integration SI_1 with components $C = c_1, \ldots, c_{10}$, the set of steps $S = \{s_1, \ldots, s_{10}\}$ and the mappings $m(1) = \{c_1, \ldots, c_9\}, m(2) = \{c_{10}\}, m(3) = \emptyset, \ldots, m(10) = \emptyset$. The testability cost CT_T of the system under integration SI_1 is then

$$CT_T(SI_2) = \sum_{s \in S} \frac{|m_C(s)|^2}{|C|^2} = \frac{9^2}{10^2} + \frac{1^2}{10^2} = 0.82$$

The schedule from above has a nine out of ten components assigned to a single integration set, step s_1 in this case. The tenth component is integrated in the second step. The testability cost metric could be easily lowered by using one additional integration step. An alternative System under integration SI_2 with significantly lower testability cost has the following component-to-step mapping: $m(1) = \{c_1, \ldots, c_4\}, m(2) = \{c_5, c_6, c_7\}, m(3) = \{c_8, c_9, c_{10}\}, m(4) = \emptyset, \ldots, m(10) = \emptyset$. The testability cost CT_T of the system under integration SI_2 is then

$$CT_T(SI_2) = \sum_{s \in S} \frac{|m_C(s)|^2}{|C|^2} = \frac{4^2}{10^2} + \frac{3^2}{10^2} + \frac{3^2}{10^2} = 0.34$$

Potential Extensions: The testability metric in its current form completely ignores the diversity of the components, it regards the test effort

as equal among all components. An extended version of the testability metric takes the complexity of the component interconnections into account. A first step would be to include the number of ports of each component in the metric. Another aspect would be to include the safety relevance of the component as a factor of influence.

Stub Development

It is necessary to satisfy a component's requirements, like data flow or control flow dependencies, for the components to be integrated. If the dependencies are not satisfied, the integrator has to implement stubs to perform a successful integration. With each stub the test effort for the particular integration step rises, since the stubs have to be tested as well. In addition, the use of stubs carries the risk of duplicate implementations in cases where the actual component is integrated and the stub has not been removed yet. Also, it is necessary to re-test components which were enabled to be integrated by stubbing of their dependencies. This can be done by regression testing with a component that was previously stubbed.

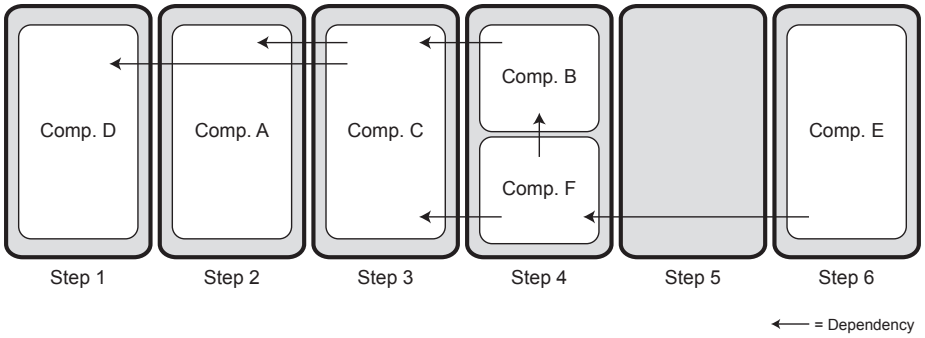


Figure 7.4: An optimal integration sequence in respect to stub development cost. The arrows represent dependency relations between the components. In this sequence no dependency relation remains unsatisfied at any integration step. Therefore, no stubs have to be introduced.

The cost for unsatisfied dependencies CT_D is

$$CT_D(SI) = \sum_{c \in C} |m_D(c)_{post}| \text{, with}$$
$$m_D(c)_{post} = \{c_D \in m_D(c) | m_S(m_C^{-1}(c)) < m_S(m_C^{-1}(c_D))\}$$

with the set of components $m_D(c) = C \times C$ on which the component c depends and the set of components $m_D(c)_{post}$ on which the dependencies of c are unsatisfied. This set consists of all components c_D, which are part

of an integration step $m_S(m_C^{-1}(c_D))$ later in the sequence than the step of component c.

Component Availability Violation

There are two points in time that represent a timeframe for the integration of a particular component. This timeframe spans from the completion of the components implementation phase (including module test) to its release date. The timeframe of a components availability poses a restriction on the ability to integrate a particular component. If this restriction is violated, the integrator has to invest additional effort to complete the integration sequence. If the desired integration plan intends to integrate a component before it becomes available it is necessary to implement a stub or force an advanced deployment of the component.

In the second case, the integration plan violates the delivery date of the component. The integrator and the software project itself have to face contractual penalties from their customers because of the violated release date. The cost for a violation of the availability and delivery limit CT_A is

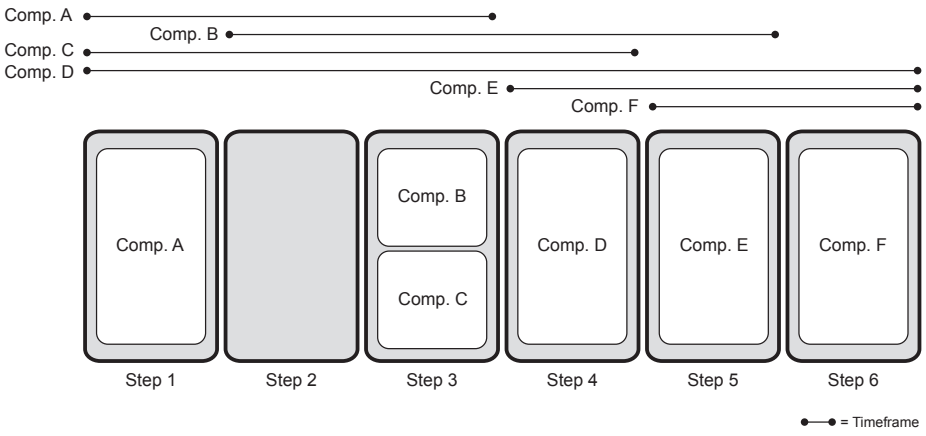


Figure 7.5: An optimal integration sequence concerning the components' availability timeframes. The timeframes are represented by the circle-headed lines above the diagram. All components are assigned to integration steps that are enclosed in their availability timeframe.

$$CT_A(SI) = \sum_{c \in C} \frac{f(c)}{max(f(m_C(c)))}$$

$$f(c) = \begin{cases} 0, & \text{if } m_A(c) \cap m_C(c) \neq \emptyset\,, \\ k_A^{start}(min(m_S(m_A(c))) & \text{if } m_A(c) \cap m_C(c) = \emptyset \wedge \\ -m_S(m_C(c))), & \quad \forall i \in m_S(m_A(c)) : \\ & \quad m_S(m_C(c)) < i\,, \\ k_A^{end}(m_S(m_C(c)) & \text{if } m_A(c) \cap m_C(c) = \emptyset \wedge \\ -max(m_S(m_A(c)))), & \quad \forall i \in m_S(m_A(c)) : \\ & \quad m_S(m_C(c)) > i\,. \end{cases}$$

The indices of the boundary steps of the availability timeframe of the component c are $min(m_S(m_A(c)))$ and $max(m_S(m_A(c)))$. The fixed costs for the violation of the corresponding limits per single step are k_A^{start} and k_A^{end}.

Human Resource Availability Violation

The influence on the integration cost of the availability of human resources during the integration phase is similar to the availability and delivery of the components itself. It is possible to improve the testability when parallel integration is possible. This is the case, if the integrated components are independent. If human resources are dedicated to the project but are not used, these resources can be considered as wasted and will assign a cost penalty in this case. It may also be the case that certain components need a particular specialist for integration. It restricts the possible strategies if this is a hard requirement, or it will impose a penalty on every violation of this assignment.

The cost for the human resource mapping is

$$CT_H(SI) = \sum_{s \in S} CT_{H,unused}(s) + CT_{H,unavailable}(s)$$

$$CT_{H,unused}(s) = \begin{cases} 0, & \text{if } m_C^{-1}(s) \neq \emptyset, \\ m_R(s), & \text{if } m_C^{-1}(s) = \emptyset. \end{cases}$$

$$CT_{H,unavailable}(s) = \begin{cases} 0, & \text{if } m_R(s) \neq \emptyset, \\ m_C^{-1}(s), & \text{if } m_R(s) = \emptyset. \end{cases}$$

with the cost for unused human resources $CT_{H,unused}$, the number of assigned resources in step s is $m_R(s)$, the cost for needed but unavailable resources $CT_{H,unavailable}$.

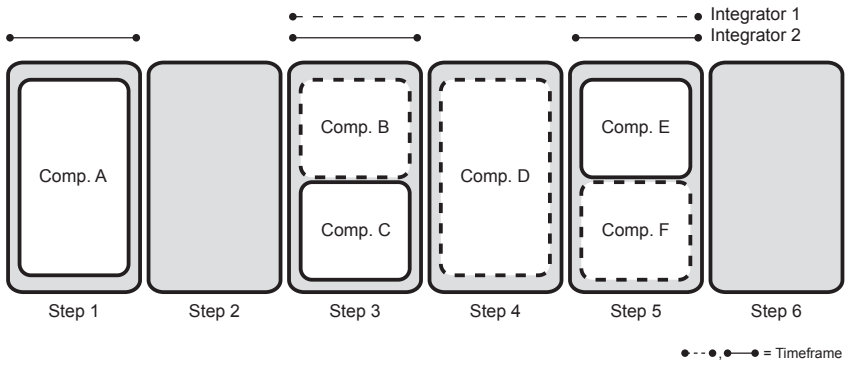


Figure 7.6: An optimal integration sequence concerning the human resource availability timeframes. The timeframes are represented by the circle-headed lines above the diagram. The relation between resources and components is shown by their designated line styles. In this sequence, the components are integrated only in integration steps in which their required resource is available.

Summary

The accumulated cost $CT(SI)$ for system under integration is defined as

$$\begin{aligned} CT(SI) =& CT_L(SI) + CT_T(SI) + \\ &+ CT_D(SI) + CT_A(SI) + CT_H(SI) \ , \end{aligned}$$

with the system under integration SI, cost for the temporal length of integration sequence CT_L, testability cost CT_T, dependency cost CT_D, the component availability cost CT_F and human resource cost CT_H.

7.3 Integration Planning Problem

The cost measurement method from the preceding section makes an individual rating of integration sequences possible. The formalization of the so-called integration planning problem shall give the integrators decision support on how to design such an integration sequence in the beginning.

Finding the optimal integration sequence can be described as a classical optimization problem, e.g. by finding minima or maxima of a particular function:

$$\begin{aligned} f : A &\rightarrow \mathbb{R} \\ \min &\left(f(x) | x \in A\right) \end{aligned} \tag{7.1}$$

In this case, the objective function seeks the minimum of $f(x)$ for the value of x in a solution space $A \subseteq \mathbb{R}$. The main problem of this optimization problem emerge from the particular context of the integration planning problem - the size of its solution space and the complexity of its objective function.

7.3.1 Solution Space

The set of possible integration sequences for n components on m steps are defined as Seq(n, m). To compute the contents of Seq(n, m), every ordered partition of C has to be taken into account to calculate all possible integration sequences. In addition to that, empty integration steps (gaps) can exist between other integration steps. This is the case if the component are unevenly distributed among the possible integration steps or if there are simply more possible integration steps than components. Also, the size of the solution space depends on the maximum number of possible integration steps. The solution space for an integration sequence of n components onto m possible integration steps is defined as:

$$|\text{Seq}(n,m)| = \begin{cases} 0, & \text{if } m \leq 0, \\ \sum_{k=0}^{n} \frac{m!}{(m-k)!} \left\{ {n \atop k} \right\}, & \text{if } n \leq m, m \geq 0, \\ \sum_{k=0}^{m} \frac{m!}{(m-k)!} \left\{ {n \atop k} \right\}, & \text{if } n > m, m \geq 0. \end{cases} \tag{7.2}$$

The term $\left\{ {n \atop k} \right\}$ denotes the Stirling numbers of the second kind. The Stirling number of the second kind are defined as

$$\left\{ {n \atop k} \right\} = \frac{1}{k!} \sum_{j=0}^{k} (-1)^{k-j} \binom{k}{j} j^n. \tag{7.3}$$

They count the number of possible ways to partition a set of n elements into k non-empty subsets.

Table 7.1 shows exemplary values of $|Seq(n, m)|$ of the number of possible integration sequences.

Because the integration of component-based systems in the automotive industry can consist of integrating up to 1000 components, finding the optimal strategy through minimization of the cost function from equation 7.4 will become practically impossible to complete with exhaustive search, due to spatial and temporal restrictions. In the following, the objective function for the integration problem its formulated and possible solution methods are presented.

Table 7.1: Exemplary values for the number of possible integration sequences $|Seq(m,n)|$

n	m	$\|Seq(n,m)\|$
5	5	3125
10	10	1.00×10^{10}
20	20	$\approx 8.27 \times 10^{20}$
50	50	$\approx 8,88 \times 10^{84}$
100	100	$> 10^{200}$

7.3.2 Integration Planning Problem Objective Function

In the case of the integration planning problem the following objective function applies for a system under integration SI[2]:

$$\begin{aligned} CT &: \text{Seq}(C,S) \to \mathbb{R} \\ \min &\left(CT(m_C) | m_C \in \text{Seq}(|C|,|S|)\right) \\ \text{with } &m_D, m_R, m_{RC} \text{ and } m_A \text{as conditions.} \end{aligned} \tag{7.4}$$

The objective function searches the mapping m_C of components to integration steps with the lowest integration cost $CT(m_C)$. The solution space consist of the set of possible integrations for SI, which given by Seq(C,S). The objective function is restricted by the conditions m_D, m_R, m_{RC} and m_A. They represent the dependencies between the components, the resource availability during the integration, the mapping of resources to components and the availability of the components.

7.3.3 Solving the Integration Planning Problem

Section 7.3.1 illustrated that the solution space of the integration planning problem is growing more than exponential with the number of components in the integration sequence. Also, section 7.2 showed that measuring the total cost of integration, which is the target of optimization, can only be described by dividing it into separate cost measurements for individual aspects of the integration sequence.

[2]in the form of $SI = (C, S, R, m_C, m_S, m_D, m_R, m_{RC}, m_A)$, see definition 9 of section 7.2.1

7.3.4 Related Work

There are established optimization methods which are applicable to comparable optimization problems. In the following, a selection of suitable optimization methods is discussed. A comprehensive overview of such techniques can be found in Edelkamp and Schrödl, 2012.

Simulated Annealing was proposed first in Kirkpatrick, Gelatt, and Vecchi, 1983. The method is founded on the physical process of annealing and leverages statistical mechanics to perform an iterative search of the optimum in the solution space. Simulated annealing interprets the objective function of an optimization problem as the energy of thermodynamic system. The main concept of simulated annealing consists of the probabilistic acceptance of a solution the exploration of the solution's neighborhood and a cooling schedule that respects the thermal equilibrium. The optimal solution is found through consequently lowering the temperature in this system. The temperature represents the probability to select a solution during an iteration with a higher energy. This enables simulated annealing algorithms to escape local, false minima.

Genetic Algorithms mimic the process of natural evolution. They were introduced first in Holland, 1975. They implement an abstracted version of the concepts of selection, mutation, recombination and fitness. A basic genetic algorithm contains an initial population that consists of initial individuals that represent potential problem solutions. The fitness function correlates with the objective function of the optimization problem and describes the quality of a solution. The selection function selects individuals from the population according to the fitness function. The recombination function combines two selected solutions to a new solution in the population. During the recombination process there mutations occur with a predefined probability. The genetic algorithm terminates when the termination criteria is met, e.g. the fitness function does not improve over a selected number of generations.

There are other methods, e.g. Particle Swarm Optimization (cf. Kennedy and Eberhart, 1995), Ant Algorithms (Doringo1992) or Lagrange Multipliers (cf. Avriel, 2003), which could be promising options but will not be explained in detail here.

The optimizations methods from above are well proven in various problem scenarios and they present a comprehensible choice in the integration planning problem. The work on interface automata and especially their verification method (cf. chapter 6) suggests that the field of game theory and its application to optimization problems is also promising a successful solution method to the integration planning problem. Research work in this field is often carried out with focus to economic

problems like market and auction optimization. The publications in context of computer science are mainly fundamental work that focuses on mathematical properties of games and their representations as well as the existence and feasibility of solution concepts. Extensive practical applications, tool chains or APIs are rare and make this approach more challenging. The following section describes the basics of game theory, related work in using game theory for optimization problems and the concept of integration games.

7.4 Game Theory

The field of game theory offers numerous solution concepts for a wide range of problems, including optimization problems. All game models have the following definition in common: They describe a situation in which a set of players with conflicting interests compete for gains. Every decision of the players is associated with a payoff.

A very common example in the field of game theory is the prisoners dilemma (cf. Luce and Raiffa, 1957). It has the following decision situation:

> Two suspects are taken into custody. The state attorney is convinced that they are guilty in the case of a capital crime but he does not have enough evidence to charge them in court. He informs the two suspects that they have two possibilities: to testify or to refuse to testify. If both suspects refuse to testify, the state attorney will press charges for minor offenses like gun possession and they will get a small sentence. If both testify, they will be charged together but not with the maximum sentence. If one suspect testifies and the other doesn't, the confessed offender will be released shortly but the other will get the maximum sentence.

The game situation of the two prisoners can be described formally as follows. Both have two pure strategies as players $i = 1$ or $i = 2$ ($i = (1, 2)$) *Don't testify* (s_{i1}) or *Testify* (s_{i2}). A strategy combination $s = (s_1, s_2)$ results depending on which strategy the players choose. Table 7.2 shows the stately combinations of the prisoners dilemma and the corresponding payoffs for the two players.

A strategic game $\Gamma = (N, S, u)$ is defined as (cf. Holler and Illing, 2006):

1. the set of players $N = 1, \ldots, n$,
2. the strategy space S, which contains the set of all possible strategy combinations $s = (s_1, \ldots, s_i, \ldots, s_n)$ of the players' strategies, i.e. $s \in S$

Table 7.2: Prisoner's Dilemma

	s_{21}	s_{22}
s_{11}	1,1	10,0
s_{12}	0,10	8,8

3. the payoff functions $u = (u_1, \ldots, u_n)$. $u_i(s)$ is the payoff for player i when the strategy s is played. The function $u_i()$ is called payoff function.

4. the game rules (as far the are determined by the strategy space S_i)

If a specific strategy combination s is played in a game $\Gamma = (N, S, u)$, the result in payoff combination is $u(s)$. The set of all possible payoff combinations, the payoff space is denoted as:

$$P = \{u(s)|s \in S\} = \{(u_1(s), \ldots, u_n(s)) \text{ for all } s \in S\}.$$

7.4.1 Solution Concepts and Equilibria

In the case of the prisoner's dilemma the solution can be found through the concept of dominance between the strategies. The *Don't Testify* strategy is dominated by *Testify* for every strategy combination. If player 2 selects strategy s_{21}, then player 1 receives a higher payoff if he selects s_{12}. However, if player 2 testifies, the strategy s_{12} still is the best strategy for player 1. The decision of player 1 is therefore independent of the decision of player 2. The same applies also from the viewpoint of player 2. The strategy *Testify* is the dominant strategy for both players: $u_i(s_{i2}, sik) > u_i(s_{i1}, s_{ik})$.

The solution of games in the game-theoretic sense is the search for a strategy combination in which none of the involved players will deviate from their decision. Such a solution or strategy combination is called Nash equilibrium. Each equilibrium of dominated strategies is also a nash equilibrium. Note that games can contain more than one Nash equilibrium.

7.4.2 Application to the Integration Planning Problem

The approach, which is presented in this work interprets the search for an optimal integration strategy as a competition between the components over the *best* spot in the integration schedule. The components assign themselves to the integration schedule's integration steps. This assignment is based on the players payoff for choosing a particular integration

step. The solution of such a game will yield a nash equilibrium, which represents a state of the game, in which none of the players will change their decision to alter the game outcome. In the case of integration modeling this means that the components have found a stable integration sequence, where no component will change its chosen integration step.

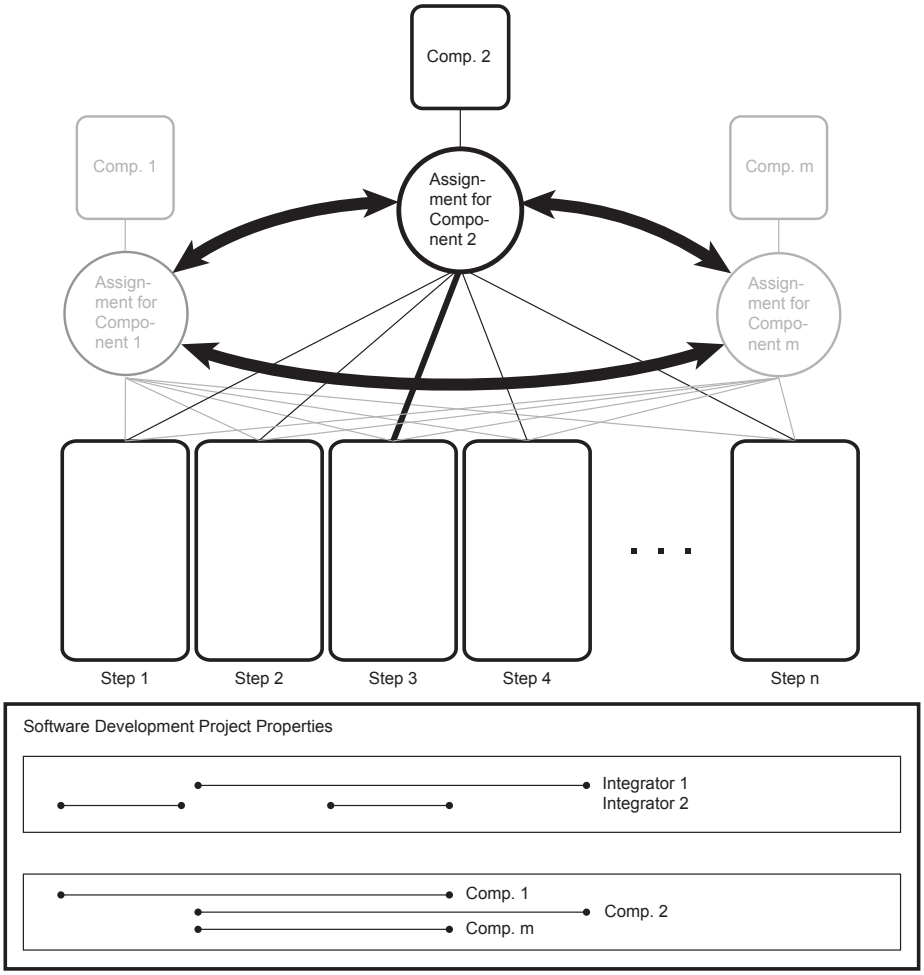


Figure 7.7: Integration Game Concept. A number of components (three components in this case) have to choose an assignment to a set of integration steps. Their assignment depends on the assignment decisions of the other involved components and is also influenced by the properties of the software development project.

Similar work was done in Gąsior and Drwal, 2012, where a allocation game for optimizing the network utility in internet networks was presented. Sim, Lee, and Kim, 2004 presented an more general approach to model multi-objective optimization problems with use of a co-evolutionary solution. The field of operations research also uses game theoretic con-

cepts. In Cachon and Netessine, 2006, applications of game theory in supply chain analysis is presented.

7.4.3 Selection of the Game Representation

The central approach to solve the integration planning problem is a definition of a game between the components. These games are described in so-called game representations. These representations have been developed with aim to very different problem sets. Therefore, it is necessary to choose a suitable game representations for the integration planning problem. Important for the selection of the game representation is the overall possibility to model the problem and the effort to describe the game with the particular representation. The following includes a description of common game representation formalisms like the normal-form and the extensive-form representation as well as more advanced formalisms like graphical games and potential games. The properties and advantages of the action-graph game representation formalism that was used to model the integration game are discussed afterwards.

Normal-form Representation

The normal-form representation for games consists of a table of payoff entries. For a two-player game like the example in table 7.3, the columns represent the actions, which can be taken by the first player whereas the rows represent the second player's actions. The payoff for each combination of actions is a comma-separated value, where the first value being the payoff for the first player and the second value being the payoff for the second player respectively. It is necessary to introduce paginated table views and nested tables to describe games with more than two players in the normal form.

Table 7.3: Example two-player game in normal form.

	A	B
A'	0, 0	2, 1
B'	1, 2	3, 1

To describe the integration problem in the normal form, it is necessary to construct a table with dimensions equal to the number of components to be integrated. In addition to that, the number of cells in that table is equal to the number of ordered partitions (see equation 7.3). This means the description of such a game in the normal form is equally complex as

the exhaustive search for the integration sequence with the lowest overall cost in the set of all integration sequences.

Extensive Representation

Extensive-form game representations are tree structures, which show the players' decisions as nodes and the actions as edges. The leaves of the tree represent the final state of the game and the according payoffs. Figure 7.8 shows an example of a game in extensive form.

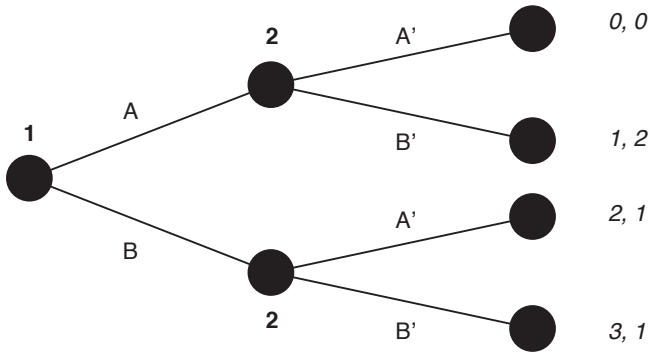


Figure 7.8: Example game in extensive form.

Compared to the normal-form representation the extensive-form representation has a slightly increased representation size.

Graphical Games

Graphical games were introduced by Kearns in Kearns, Littman, and Singh, 2001. An example graphical game is illustrated in figure 7.9. The nodes represent the players in the game and the edges represent the relations between the players. Players, whose utility functions are affected by other players are connected through edges. It is possible to represent all games that have an normal form representation as graphical game. The solution of the game can be computed depending on the graphical representation rather than the representation of the induced normal form. Graphical games do not take advantage of anonymity, the players utilities only depend on the number of players who took a specific action.

Figure 7.9: Example game as graphical game. The nodes represent the players in the game. The edges represent relationships between the utility functions of the players.

Congestion and Potential Games

Congestion games were introduced by Monderer et al. in Monderer and Shapley, 1996. Congestion games take advantage of symmetry, anonymity and context-specific independencies. Figure 7.10 shows an example game, represented as congestion game. The nodes represent resources or facilities. The circles represent the actions that the particular players can take. Congestion games always have pure-strategy equilibria. Therefore, they are not able to represent games that do not have pure-strategy equilibria.

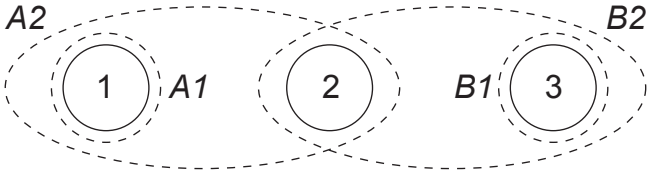


Figure 7.10: Example game as congestion game.

Action-Graph Games

Action-graph games (AGG) were introduced by Xin Jiang et al. in Jiang, Leyton-Brown, and N. Bhat, 2010. They combine the advantages of graphical and congestion games. It is possible to represent any game as an action graph. Action-graph games take advantage of symmetry, anonymity and context-specific independencies to model complex games very compact. Action graph games promise a solution to describe the complex nature of the integration planning problem.

Action-Graphs are the central model for action graph games. They are defined as follows Jiang, Leyton-Brown, and N. Bhat, 2010: An action graph $G = (\mathcal{A}, E)$ is a directed graph where:

- $\mathcal{A}$ is the set of nodes. We call each node $\alpha \in \mathcal{A}$ an action, and $\mathcal{A}$ the set of distinct actions. For each agent $i \in N$, let A_i be the set of actions available to i, with $\mathcal{A} = \bigcup_{S_i \in N} A_i$. We denote by $a_i \in A_i$ one of agent i's actions. An action profile (or pure strategy profile) is a tuple $a = (a_1, \ldots, a_n)$. Denote by A the set of action profiles. Then $A = \prod_{i \in N} A_i$ where $\prod$ is the Cartesian product.

- E is a set of directed edges, where self edges are allowed. We say α' is a neighbor of α if there is an edge from α' to α, i.e., $(\alpha', \alpha) \in E$. Let the neighborhood of α, denoted $v(\alpha)$, be the set of neighbors of α, i.e., $v(\alpha) \equiv \{\alpha' \in \mathcal{A} | (\alpha', \alpha) \in E\}$.

According to Jiang, Leyton-Brown, and N. Bhat, 2010, action-graph games are defined as: An action-graph game is a tuple (N, A, G, u) where

- N is the set of players;
- $A = \prod_{i \in N} A_i$ is the set of action profiles;
- $G = (\mathcal{A}, E)$ is an action graph, where $\mathcal{A} = \bigcup_{i \in N} A_i$ is the set of distinct actions;
- u is a tuple $(u^{(\alpha)})_{\alpha \in \mathcal{A}}$, where each $u^{\alpha} : C^{(\alpha)} \mapsto R$ is the utility function for action α. Semantically, $u^{\alpha}(c^{(\alpha)})$ is the utility of an player who chose α, when the configuration over $v(\alpha)$ is $c^{(\alpha)}$.

For notational convenience, we define $u(\alpha, c^{(\alpha)}) \equiv u^{\alpha}(c^{(\alpha)})$ and $u_i(a) \equiv u(a_i, \mathcal{C}^{(a_i)}(a))$. $A_{-i} \equiv \prod_{j \neq i} A_j$ is the set of action profiles of players other than i, and denote an element of A_{-i} by a_{-i}.

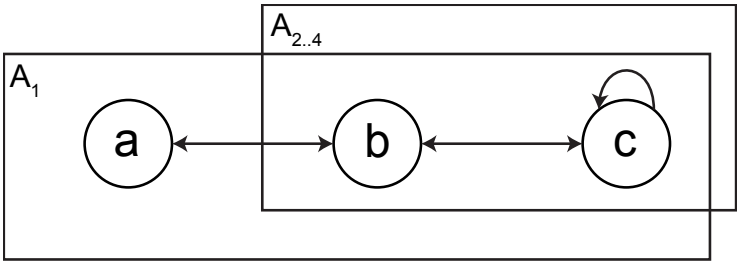


Figure 7.11: An example action-graph game with following properties: $N = \{1, 2, 3, 4\}; \mathcal{A} = \{a, b, c\}; A_1 = \{a, b, c\}; A_{2..4} = \{b, c\}; u(c) = c - b^2$

In order to model complex games with an action-graph game, it possible to extend the action-graph with so-called function nodes (AGGFN). Note that every action-graph game with function nodes can be mapped to a corresponding action-graph game, the so-called induced AGG of the AGGFN. The action-graph game with function nodes is defined in Jiang, Leyton-Brown, and N. Bhat, 2010 as: An Action-Graph Game with Function Nodes (AGGFN) is a tuple $(N, A, \mathcal{P}, G, f, u)$, where:

- N is the set of players;
- $A = \prod_{i \in N} A_i$ is the set of action profiles;
- $\mathcal{P}$ is a finite set of function nodes;
- $G = (\mathcal{A} \cup \mathcal{P}, E)$ is an action graph, where $A = \bigcup_{i \in N} A_i$ is the set of distinct actions. The restriction of G to the nodes $\mathcal{P}$ is required to be acyclic and for every $p \in \mathcal{P}$ there exists an $m \in \mathcal{A} \cup \mathcal{P}$ such that $(p, m) \in E$;
- f is a tuple $(f^p)_{p \in \mathcal{P}}$, where each $f^p : C^{(p)} \mapsto \mathbb{R}$ is an arbitrary mapping from neighbors of p to real numbers;

- u is a tuple $(u^\alpha)_{\alpha \in \mathcal{A}}$, where each $u^\alpha : C^{(\alpha)} \mapsto \mathbb{R}$ is the utility function for action α.

A function node p of a AGGFN holds an arbitrary function that maps a configuration of the p's incoming neighbors to real numbers. The utility function of action nodes can be defined to produce payoffs with respect to the function node's result. An example for a particular function node type is given in Jiang, Leyton-Brown, and N. Bhat, 2010. The so-called simple aggregator is defined as:

A function node $p \in \mathcal{P}$ is a simple aggregator if each of its neighbors $v(p)$ are action nodes and f^p is the summation function: $f^p(c^{(p)}) = \sum_{m \in (p)} c(m)$.

Simple aggregator function nodes takes the value of the total number of players who chose any of the node's neighbors. Figure 7.12 shows an example of an action graph game with a function node. The function node p_1 aggregates the number of players choosing either action a or action b and reduces that number by one. The payoff function of action node c can then include the result of the function node p_1 in its calculation.

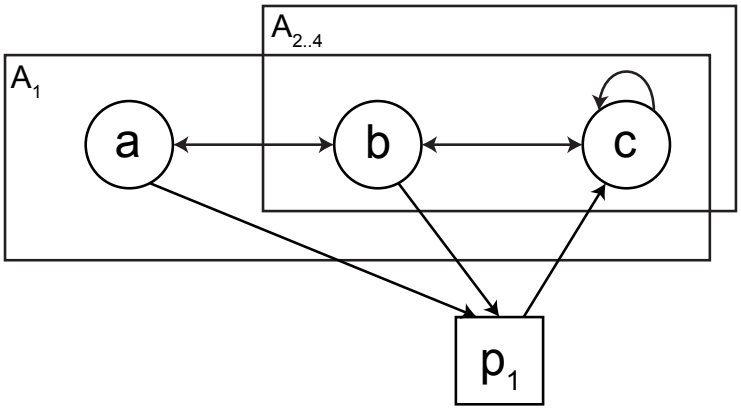


Figure 7.12: An example action-graph game with function nodes with following properties:
$N = \{1, 2, 3, 4\}; \mathcal{A} = \{a, b, c\}; \mathcal{P} = \{p_1\}; A_1 = \{a, b, c\}; A_{2..4} = \{b, c\}$; An example payoff function for players, choosing action c is $u(c) = c - b^2 + f_1$. The function node p_1 has the following function $f^p(c^{(p)}) = \left(\sum_{m \in (p)} c(m)\right) - 1$.

7.5 Integration Game Model

This section defines a model of the integration game with use of the action-graph game formalism. This model is called the Integration Game Model

(IGM).

The IGM is built by successively adding the integration cost types (see 7.2) to an initial model. This illustrates the effects of the single optimization dimensions on the complexity of the game. The initial model considers only the testability of the system. Table 7.4 shows the four stages of the IGM, which are described in this work.

Table 7.4: The different stages of the IGM.

Abbrev.	Description
Stage 1	This stage only takes the length of the sequence and the testability of the system into account.
Stage 2	This stage takes the length of the sequence, testability and fixed component timeframes into account.
Stage 3	This stage takes the length of the sequence, testability, component timeframes and dependencies between components into account.
Stage 4	This stage is similar to the third stage. Additionally, it allows a violation of the component availability timeframe but allocates penalties to the payoffs in this case.

The following sections describe the stages from 7.4 in detail.

The general concept of the IGM is to model a game between the components, which compete for places in the integration schedule. Their decisions in this game are annotated with payoffs, which are derived from the costs function of the integration (cf. section 7.2). The components are therefore competing for their own benefit but are indirectly optimizing the integration cost function. Every configuration of an action node has a dedicated payoff. The configurations are the set of all possible chosen or not chosen neighbor action nodes that influence the particular action node.

7.5.1 Stage 1: Sequence Length and Testability

The first stage of the IGM focuses only on two aspects of the integration costs (see section 7.2). These are the overall length of the integration, which should be as short as possible, and the testability of the system, which is decreasing with number of components per integration step.

A minimal example of the stage 1 game is shown in figure 7.13. The integration steps are represented as action nodes and their number is equal to the number of components in the system. They are ordered in a

sequence from left to right. The leftmost action node represents the first possible integration step, the rightmost action node is the last integration step. The number of players in this game is equal to the number of components in the system. All components - the players - are able to select an arbitrary integration step. Therefore, they share a single action set, which includes all action nodes.

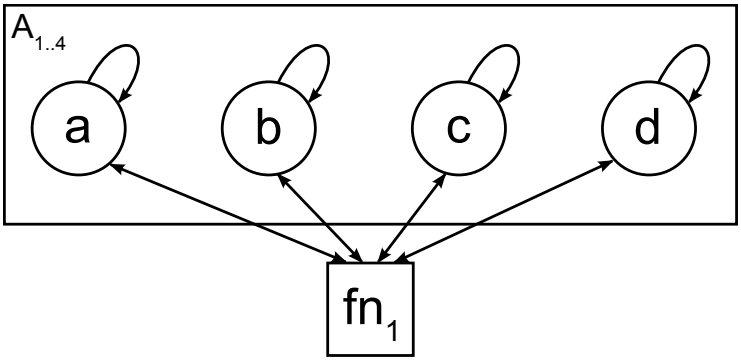


Figure 7.13: An example game in stage 1.

Every action node has an edge pointing to itself to model the testability aspect of the integration costs. Through this it is possible to assign payoffs depending on the number of components, which choose the same integration step.

Stage 1 also includes a function node. This function node is bidirectionally connected to all action nodes. It is of the type 'highest index of non-zero node', which means that it is now possible to assign payoffs, depending on which is the highest index of all taken actions. The function node is used to model the sequence length cost of the integration costs from section 7.2.2. In the stage 1 model and in all other stages, the indices of the action nodes are increasing from left to right.

Payoff Formula

The payoff formula for a component in a particular integration step is the cost for the integration length plus the cost for testing of the component which chooses this particular action node in each configuration. The payoffs are negative since the optimal solution has minimal integration costs.

$$\begin{aligned} P^{conf}(a) &= -(P_L^{conf} + P_T^{conf}) \\ &= -(k_L \frac{fn_{length}}{|\mathcal{A}|} + k_T \frac{a}{|N|}) \end{aligned}$$

Example Action Graph Game

The example action graph game $AGGFN_1 = (N, A, \mathcal{P}, G, f, u)$ of stage 1 from figure 7.13 has the following properties: A set of players $N = \{1, 2, 3, 4\}$, the action sets $A_{1..4} = \{a, b, c, d\}$. The action graph $G = AG_1$, the set of functions $f = \{f^{fn_1}\}$ with $f^{fn_1} = max_{c(v(fn_1))}$ and finally the utilities $u = \{u^a, u^b, u^c, u^d\}$ with the functions

$$u^a = -\left(\frac{fn_1}{|\mathcal{A}|} + \frac{v(a)}{|N|}\right), u^b = -\left(\frac{fn_1}{|\mathcal{A}|} + \frac{v(b)}{|N|}\right),$$
$$u^c = -\left(\frac{fn_1}{|\mathcal{A}|} + \frac{v(c)}{|N|}\right) \text{ and } u^d = -\left(\frac{fn_1}{|\mathcal{A}|} + \frac{v(d)}{|N|}\right).$$

The action graph $AG_1 = \{\mathcal{A}, E\}$ of the first stage example consists of the actions $\mathcal{A} = \{a, b, c, d\}$, the function nodes $\mathcal{P} = \{fn_1\}$ the edges

$$E = \{(a, a), (b, b), (c, c), (d, d), (a, fn_1), (b, fn_1),$$
$$(c, fn_1), (d, fn_1), (fn_1, a), (fn_1, b), (fn_1, c), (fn_1, d)\}.$$

7.5.2 Stage 2: Component Timeframes

The second stage adds another aspect of the integration cost to the model. The timeframe, during which the components becomes available and need to be delivered is added to the model. This means, the action sets of the components are changed corresponding to their availability. Figure 7.14 shows an example of the model. In this example, the components which are represented as *Player 3* and *Player 4,* can only choose the third and the fourth action node whereas *Player 1* and *Player 2* are not restricted in their decision.

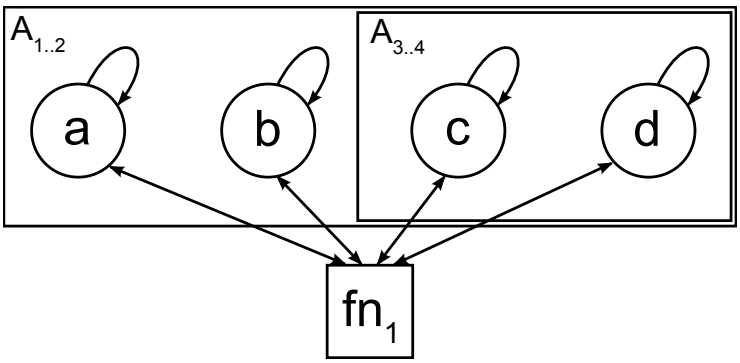


Figure 7.14: An example game in stage 2.

Payoff Formula

Only the players' action sets are changed, so the payoff formula for each player is equal to the first stage. This restriction will lead to very distinct solutions when compared to stage 1.

$$\begin{aligned} P^{conf}(a) = & -(P_L^{conf} + P_T^{conf}) \\ = & -(k_L \frac{fn_{length}}{|\mathcal{A}|} + k_T \frac{a}{|N|}) \end{aligned}$$

Example Action Graph Game

The example action graph game $AGGFN_2 = (N, A, \mathcal{P}, G, f, u)$ of stage 2 from figure 7.14 has the following properties: A set of players $N = \{1, 2, 3, 4\}$, the action sets $A_{1..2} = \{a, b, c, d\}$ and $A_{3..4} = \{c, d\}$. The action graph $G = AG_2$, the set of functions $f = \{f^{fn_1}\}$ with $f^{fn_1} = max_{c(v(fn_1))}$ and finally the utilities $u = \{u^a, u^b, u^c, u^d\}$ with the functions

$$u^a = -\left(\frac{fn_1}{|\mathcal{A}|} + \frac{v(a)}{|N|}\right), u^b = -\left(\frac{fn_1}{|\mathcal{A}|} + \frac{v(b)}{|N|}\right),$$
$$u^c = -\left(\frac{fn_1}{|\mathcal{A}|} + \frac{v(c)}{|N|}\right) \text{ and } u^d = -\left(\frac{fn_1}{|\mathcal{A}|} + \frac{v(d)}{|N|}\right).$$

The action graph $AG_2 = \{\mathcal{A}, E\}$ of the second stage example consists of the actions $\mathcal{A} = \{a, b, c, d\}$, the function nodes $\mathcal{P} = \{fn_1\}$ the edges

$$E = \{(a, a), (b, b), (c, c), (d, d), (a, fn_1), (b, fn_1),$$
$$(c, fn_1), (d, fn_1), (fn_1, a), (fn_1, b), (fn_1, c), (fn_1, d)\}.$$

7.5.3 Stage 3: Component Dependencies

The third stage denotes a major difference to its preceding stages as it incorporates the concept of dependencies between components. Therefore, the players are no longer anonymous, which was leveraged in stage 1 and the stage 2 by having multiple players choose between shared actions. It is only possible to model dependencies between the players if they are distinguishable. This is achieved by the matrix form of this stage. The rows represent the players and the columns represent the steps, which every player can choose.

The testability needs to be implemented with a function node (from the type sum) which is connected bidirectionally to every action node in each step. It provides the number of players choosing a step to each action node of this particular step.

The dependency relations between a component j and the set of components it depends on D_j is modeled by function nodes (from the type

sum). These function nodes count the number of components from D_j which are integrated after j is integrated.

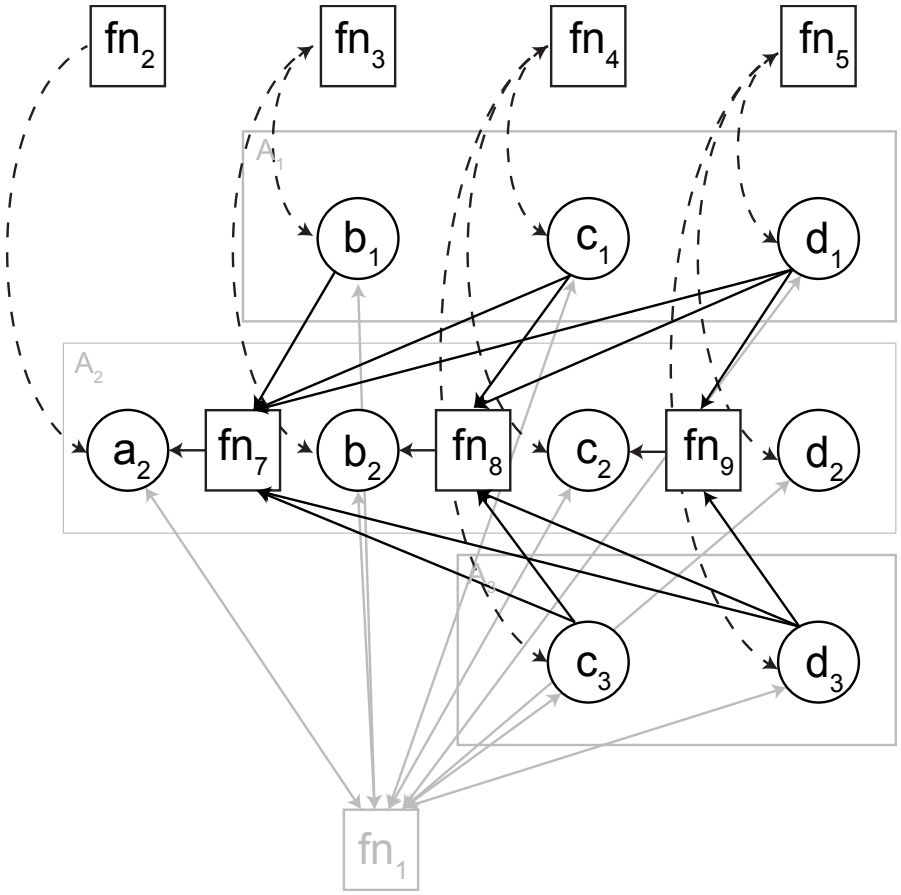


Figure 7.15: An example game in the third stage. The concept of applying action sets corresponding to the availability as well as the function node for the calculation of the sequence length are shown in light grey to enhance comprehensibility and clarity.

Payoff Formula

The payoff formula for the action nodes under a specific configuration $conf$ is:

$$\begin{aligned} P^{conf}(a) &= -\left(P_L^{conf} + P_T^{conf} + P_D^{conf}\right) \\ &= -\left(k_L \frac{fn_{length}}{|\mathcal{A}|} + k_T \frac{a}{fn_{test}(a)} + k_D fn_{dep}(a)\right) \end{aligned}$$

$f_{dep}(a)$ is the set of all function nodes that point to the action node a to model the dependencies of component a.

Example Action Graph Game

The example action graph game $AGGFN_3 = (N, A, \mathcal{P}, G, f, u)$ of stage 3 from figure 7.15 has the following properties: A set of players $N = \{1, 2, 3\}$, the action sets $A_1 = \{b_1, c_1, d_1\} A_2 = \{a_2, b_2, c_2, d_2\}$ and $A_3 = \{c_3, d_3\}$. The action graph $G = AG_3$, the set of functions

$$f = \{f^{fn_1}, f^{fn_2}, f^{fn_3}, f^{fn_4}, f^{fn_5}, f^{fn_6}, f^{fn_7}, f^{fn_8}, f^{fn_9}\}$$

with $f^{fn_1} = max_{c(v(fn_1))}$ and $f^{fn_{2..9}} = \sum_{m \in v(fn_{2..9})} c(m)$ and the utilities $u = \{u^{a_2}, u^{b_1}, u^{b_2}, u^{b_3}, u^{c_1}, u^{c_2}, u^{c_3}, u^{d_1}, u^{d_2} u^{d_3}\}$ with the functions

$$u^{a_2} = -\left(\frac{fn_1}{|\mathcal{A}|} + \frac{fn_2}{|N|} + \frac{fn_7}{|D_2|}\right), u^{b_1} = -\left(\frac{fn_1}{|\mathcal{A}|} + \frac{fn_3}{|N|}\right),$$
$$u^{b_2} = -\left(\frac{fn_1}{|\mathcal{A}|} + \frac{fn_3}{|N|} + \frac{fn_8}{|D_2|}\right), u^{c_1} = -\left(\frac{fn_1}{|\mathcal{A}|} + \frac{fn_4}{|N|}\right),$$
$$u^{c_2} = -\left(\frac{fn_1}{|\mathcal{A}|} + \frac{fn_4}{|N|} + \frac{fn_9}{|D_2|}\right), u^{c_3} = -\left(\frac{fn_1}{|\mathcal{A}|} + \frac{fn_4}{|N|}\right), u^{d_1} = -\left(\frac{fn_1}{|\mathcal{A}|} + \frac{fn_5}{|N|}\right),$$
$$u^{d_2} = -\left(\frac{fn_1}{|\mathcal{A}|} + \frac{fn_5}{|N|}\right) \text{ and } u^{d_3} = -\left(\frac{fn_1}{|\mathcal{A}|} + \frac{fn_5}{|N|}\right).$$

The action graph $AG_4 = \{\mathcal{A}, E\}$ of the third stage example consists of the actions $\mathcal{A} = \{a_2, b_1, b_2, c_1, c_2, c_3, d_1, d_2, d_3\}$, the function nodes $\mathcal{P} = \{fn_1, fn_2, fn_3, fn_4, fn_5, fn_6, fn_7, fn_8, fn_9\}$ and the edges

$$\begin{aligned} E = \{&(a_2, fn_1), (b_1, fn_1), (b_2, fn_1), (c_1, fn_1), (c_2, fn_1), (c_3, fn_1), (d_1, fn_1), (d_2, fn_1), \\ &(d_3, fn_1), (fn_1, a_2), (fn_1, b_1), (fn_1, b_2), (fn_1, c_1), (fn_1, c_2), (fn_1, c_3), (fn_1, d_1), \\ &(fn_1, d_2), (fn_1, d_3), (fn_2, a_2), (a_2, fn_2), (b_1, fn_3), (fn_3, b_1), (b_2, fn_3), (fn_3, b_2), \\ &(c_1, fn_4), (fn_4, c_1), (c_2, fn_4), (fn_4, c_2), (c_3, fn_4), (fn_4, c_3), (d_1, fn_5), (fn_5, d_1), \\ &(d_2, fn_5), (fn_5, d_2), (d_3, fn_5), (fn_5, d_3), (fn_7, a_2), (b1, fn_7), (c_1, fn_7), (d_1, fn_7), \\ &(c_3, fn_7), (d_3, fn_7), (fn_8, b_2), (c_1, fn_8), (d_1, fn_8), (c_3, fn_8), (d_3, fn_8), (fn_9, c_2), \\ &(d_1, fn_9), (d_3, fn_9)\}. \end{aligned}$$

7.5.4 Stage 4: Flexible Component Timeframes

Stage is a modification of the third stage to incorporate penalties for violation of the component availability timeframe. Stage 3 did not allow any delay or advance the integration of the particular component.

The penalties for the violation of a component's timeframe are linear increasing with the discrete distance from the latest or earliest point of the availability timeframe respectively.

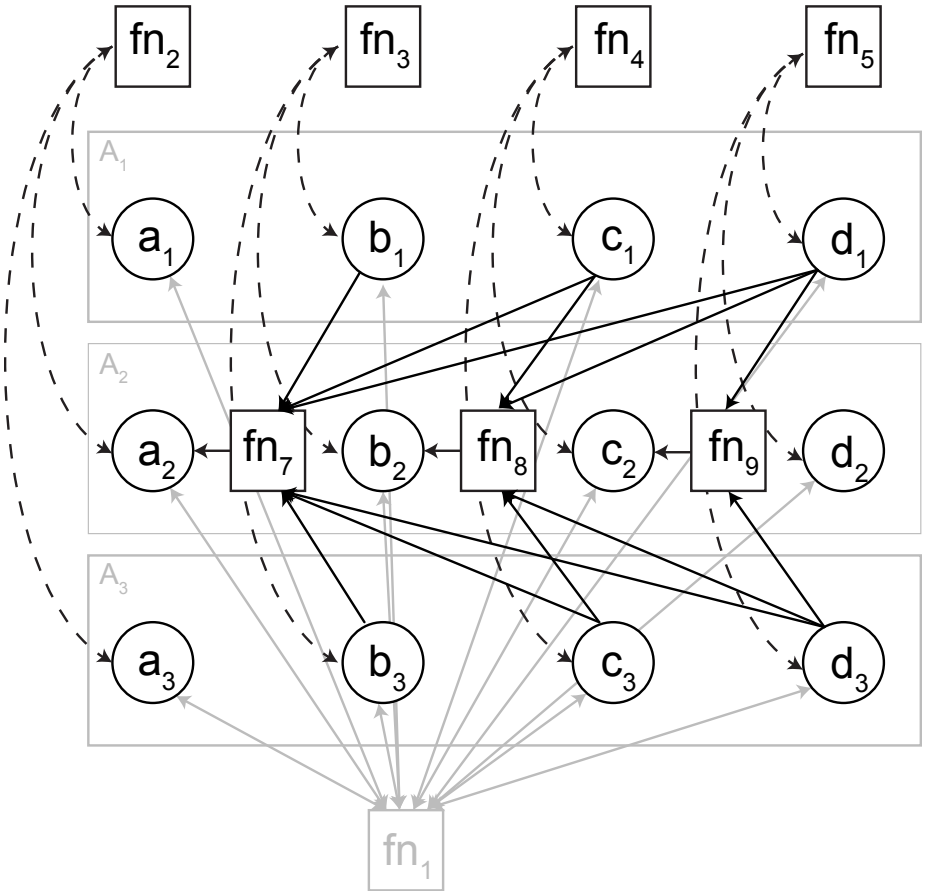


Figure 7.16: An example game in stage 4.

Payoff Formula

D_j is the set of all components on which the component j depends. $\tilde{D}_j = \{d_j \in D_J | i_{d_j} > i_j\}$ is the set of all components from D_j which are integrated after the integration step i_j of j. The fixed cost for an advanced integration of the component j is $C^F_{start,j}$, the cost for delaying the integration beyond its release date is $C^F_{end,j}$.

$$\begin{aligned} P^{conf}(a) = & -\left(P_L^{conf} + P_T^{conf} + P_D^{conf}\right) \\ = & -\Big(k_L \frac{fn_{length}}{|\mathcal{A}|} + k_T \frac{a}{fn_{test}(a)} + \\ & + k_D fn_{dep}(a) + k_F f(a)\Big) \end{aligned}$$

Example Action Graph Game

The example action graph game $AGGFN_3 = (N, A, \mathcal{P}, G, f, u)$ of stage 3 from figure 7.15 has the following properties: A set of players $N = \{1, 2, 3\}$, the action sets $A_1 = \{a_1, b_1, c_1, d_1\} A_2 = \{a_2, b_2, c_2, d_2\}$ and $A_3 = \{c_1, c_2, c_3, d_3\}$. The action graph $G = AG_4$, the set of functions

$$f = \{f^{fn_1}, f^{fn_2}, f^{fn_3}, f^{fn_4}, f^{fn_5}, f^{fn_6}, f^{fn_7}, f^{fn_8}, f^{fn_9}\}$$

with $f^{fn_1} = max_{c(v(fn_1))}$ and $f^{fn_{2..9}} = \sum_{m \in v(fn_{2..9})} c(m)$ and the utilities $u = \{u^{a_2}, u^{b_1}, u^{b_2}, u^{b_3}, u^{c_1}, u^{c_2}, u^{c_3}, u^{d_1}, u^{d_2}, u^{d_3}\}$ with the functions

$$u^{a_1} = -\left(\frac{fn_1}{|\mathcal{A}|} + \frac{fn_2}{|N|}\right), u^{a_2} = -\left(\frac{fn_1}{|\mathcal{A}|} + \frac{fn_2}{|N|} + \frac{fn_7}{|D_2|}\right), u^{a_3} = -\left(\frac{fn_1}{|\mathcal{A}|} + \frac{fn_2}{|N|}\right),$$
$$u^{b_1} = -\left(\frac{fn_1}{|\mathcal{A}|} + \frac{fn_3}{|N|}\right), u^{b_2} = -\left(\frac{fn_1}{|\mathcal{A}|} + \frac{fn_3}{|N|} + \frac{fn_8}{|D_2|}\right), u^{b_3} = -\left(\frac{fn_1}{|\mathcal{A}|} + \frac{fn_3}{|N|}\right),$$
$$u^{c_1} = -\left(\frac{fn_1}{|\mathcal{A}|} + \frac{fn_4}{|N|}\right), u^{c_2} = -\left(\frac{fn_1}{|\mathcal{A}|} + \frac{fn_4}{|N|} + \frac{fn_9}{|D_2|}\right), u^{c_3} = -\left(\frac{fn_1}{|\mathcal{A}|} + \frac{fn_4}{|N|}\right),$$
$$u^{d_1} = -\left(\frac{fn_1}{|\mathcal{A}|} + \frac{fn_5}{|N|}\right), u^{d_2} = -\left(\frac{fn_1}{|\mathcal{A}|} + \frac{fn_5}{|N|}\right) \text{ and } u^{d_3} = -\left(\frac{fn_1}{|\mathcal{A}|} + \frac{fn_5}{|N|}\right).$$

The action graph $AG_4 = \{\mathcal{A}, E\}$ of the fourth stage example consists of the actions $\mathcal{A} = \{a_2, b_1, b_2, c_1, c_2, c_3, d_1, d_2, d_3\}$, the function nodes $\mathcal{P} =$

$\{fn_1, fn_2, fn_3, fn_4, fn_5, fn_6, fn_7, fn_8, fn_9\}$ and the edges

$$E = \{(a_1, fn_1), (a_2, fn_1), (a_3, fn_1), (b_1, fn_1), (b_2, fn_1), (b_3, fn_1), (c_1, fn_1), (b_2, fn_1), (b_3, fn_1), (c_1, fn_1), (c_2, fn_1), (c_3, fn_1), (d_1, fn_1), (d_2, fn_1), (d_3, fn_1), (fn_1, a_1), (fn_1, a_2), (fn_1, a_3), (fn_1, b_1), (fn_1, b_2), (fn_1, a_2), (fn_1, c_1), (fn_1, c_2), (fn_1, c_3), (fn_1, d_1), (fn_1, d_2), (fn_1, d_3), (fn_2, a_1), (a_1, fn_2), (fn_2, a_2), (a_2, fn_2), (fn_2, a_3), (a_3, fn_2), (b_1, fn_3), (fn_3, b_1), (fn_3, b_2), (b_2, fn_3), (b_3, fn_3), (fn_2, b_3), (c_1, fn_4), (fn_4, c_1), (c_2, fn_4), (fn_4, c_2), (c_3, fn_4), (fn_4, c_3), (d_1, fn_5), (fn_5, d_1), (d_2, fn_5), (fn_5, d_2), (d_3, fn_5), (fn_5, d_3), (fn_7, a_2), (b1, fn_7), (c_1, fn_7), (d_1, fn_7), (b_3, fn_7), (c_3, fn_7), (d_3, fn_7), (fn_8, b_2), (c_1, fn_8), (d_1, fn_8), (c_3, fn_8), (d_3, fn_8), (fn_9, c_2), (d_1, fn_9), (d_3, fn_9)\}.$$

7.6 Implementation

Integration games with different complexity levels, i.e. the different development stages of the integration game model (cf. section 7.5), can be easily designed with use of the AGGUI tool Bargiacchi, Jiang, and Leyton-Brown, 2012. For a practical implementation inside of the IIS however, a extensible automated generation method is necessary. The GUI-focused AGGUI reaches its limits when it is used to produce a game file of a certain size. A specialized integration game generator (written in C++) was implemented to overcome AGGUI's limitations. This game generator accepts system definitions according to definition 9 from section 7.2.1 as input file. The integration game generator incorporates the payoff formulas from section 7.5 and produces an output file in the AGG format. The resulting game file can the be fed into the AGG/BAGG Solver Jiang, 2011. The AGG/BAGG Solver computes the game, using the Govindan-Wilson Govindan and Wilson, 2003 Global Newton Method or the Simplicial Subdivision van Der Laan et al., 1987 algorithm. The output of the solver is an example configuration of the Nash equilibrium that was found.

The quality metric for game theory-based optimization, the so called price if anarchy Koutsoupias and Papadimitriou, 1999, is defined as the ration between the optimized solution and the best case solution.

The price of anarchy for the integration game was determined by comparing the result of an exhaustive search in the problem space (see section 7.3.1) according to the metrics from section 7.2.

The experiment setup was compared with an exhaustive search for the optimal integration sequence. This search was implemented in a Maple procedure. Due to the size of the solution space, this was only possible for a maximum of six components and six possible integration steps.

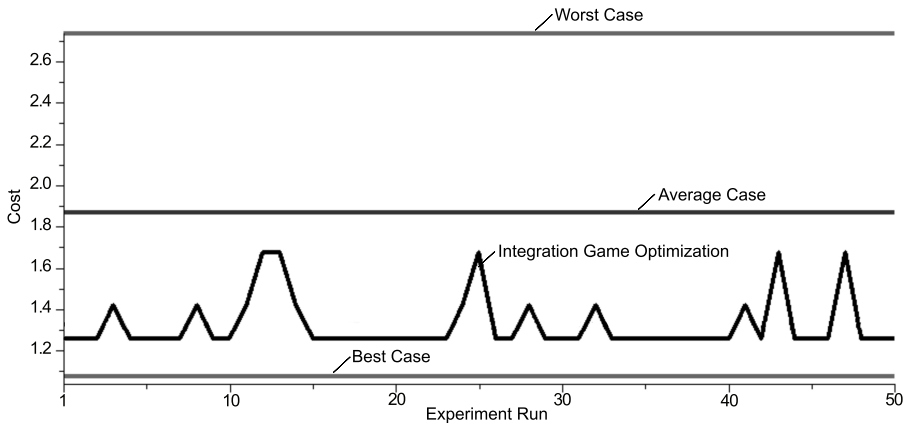


Figure 7.17: The result of the evaluation concerning the price of anarchy of the integration game.

Figure 7.17 shows how the quality of the solutions of the optimization with integration games performs in comparison to the optimal solution. In 50 runs the average quality was about 81 percent of the optimal integration sequence. This result is particularly significant when compared to the computation performance of the approach. The exhaustive search took an average of 120 seconds to compute, whereas the solution of the integration game was possible in under three seconds. When bigger and more complex models are used the exhaustive search was not able to give any results in a acceptable interval of time. Solving the integration game with a bigger model, e.g. 20 components and 20 possible integration steps, can be completed in under five minutes. The scalability of the approach is supported through the structure of the action game formalism itself. The computation of the equilibrium grows exponentially with the action graph's maximal in-degree rather than with its total number of nodes, which yields exponential savings (cf. N. A. R. Bhat and Leyton-Brown, 2004).

7.7 Conclusion

The section presented a formal model of the costs of the integration process. This model is used to formulate an optimization problem for integration scheduling. Due to the size of the solution space and also the complexity of the problem a solution is proposed, which is founded on game theory concepts. By modeling the integration game as a multiplayer game among the components with unselfish payoffs, a social optimum for the whole integration process is achieved. Additionally the

complexity of the problem required a compact game representation - in this particular case action graph games.

Chapter 8

Conclusion and Outlook

THIS thesis presented a novel approach to optimize software integration in the context of embedded automotive software. The foundation of this work lies in the virtual integration methodology, which defines a process toolset for a streamlined integration and its preparatory work to establish a efficient integration. It further described a reference architecture of the integration information system, described a method to verify compatibility between component models in the software design phase and also described a novel approach to optimize the definition of integration plans with use of game theoretic concepts.

This chapter gives a summary of the thesis and points out promising future work in this area of research.

8.1 Conclusion

The thesis presents techniques to improve the integration phase as a whole and provides means to answer the following questions:

- How can the integration phase be supported to leverage the current state of information and enable an efficient integration process?
- Is it possible to perform a verification of component behavioral compatibility in an early stage of development as a preparatory measure for integration?
- How can the integration planning process be optimized to produce cost efficient integration schedules?

It is focuses on the implementation of a reference architecture of the so-called integration information system. This information system is designed to support integrators and affiliated staff during the whole virtual

integration process. This process is divided into five parts: Compatibility verification, integration planning, integration monitoring, integration administration and build automation. Each of these process parts is reflected in the prototype implementation of the integration information system.

The thesis further describes the fundamentals and a prototypic implementation of two of the IIS's core components.

Chapter 6 presents a feasibility study of the use of interface automata to perform a compatibility check between components models. This preparatory measure helps to identify incompatible components in an early stage of development and reduces cost for error correction.

In chapter 7 the foundation of integration games is described. After the definition of a model of the integration setting and the formalization of integration cost measurement techniques the resulting optimization problem is formalized. Modeling this optimization problem with game theory techniques presents a novel approach in this field.

8.2 Further Work

The are several starting points for further work in this area. In the following section the topics are described, which improve, refine and extend the work presented in this thesis.

Case Study The reference architecture of the integration information system needs to be tested and adapted in an industry setting. A software development project has to be selected and the virtual integration methodology shall be carried out with use of the integration information system.

Performance Improvement It is necessary to optimize the solving speed for big game instances for an practical use of the approach in an industry setting. This could be accomplished by processing the solvers' matrix operations in parallel on a parallel computing platform.

Introduce Uncertainty In the model of the system under integration (cf. section 7.2.1) the attributes of the components are defined as fixed. For example, the availability of the component is set to a specific date. In a practical setting however it is more realistic that this date is a rough estimate instead of an exact value. Therefore, the optimization method must implement the element of uncertainty or incomplete information as it is called in game theory. A promising technique would be the use

of Bayesian Games as described in Harsanyi, 2004 or its compact representation, the Bayesian Action Graph Game as described in Jiang and Leyton-Brown, 2010.

Companywide Coordination of Integration Since several integration processes take place inside a company and also inside departments, a company-wide extension of the integration planning promises an effective use of human resources. This is especially important in organizations, which follow matrix management and shall help to assign human resources during the integration phase of the software products effectively.

Appendix A

Integration State of the Art Survey

Figure A.1: Integration Survey (print version, page 1)

Integration und Integrationstest von verteilten Software-Komponenten

Liebe Experten,

Ziel unseres Forschungsprojektes VitaS³ an der Hochschule Regensburg ist die virtuelle und automatisierte Integration von Softwarefunktionen in verteilten eingebetteten Automobil-Systemen.
Um praxisnah und zielgerichtet zu forschen, ist es für uns wichtig, Möglichkeiten, Herausforderungen und Grenzen bei der Integration von Softwarefunktionen zu kennen.
Ihre Erfahrungen aus der Praxis sind dabei von zentraler Bedeutung.
Daher möchten wir Sie bitten, unsere Forschung durch das Ausfüllen dieses Fragebogens zu unterstützen!

1. Name und e-mail Adresse (Angabe freiwillig)

2. In welcher Branche sind Sie tätig?

3. Wird in Ihren Projekten eine modellbasierte Entwicklung angewendet? (Wenn nein weiter zu Frage 7)
   a ☐ ja b ☐ nein c ☐ teilweise

4. Wo im Entwicklungszyklus werden Modelle genutzt?

5. Welche Modelle werden verwendet?

6. Welche Tools werden zur Modellierung verwendet?

7. Wie viele Komponenten beinhalten Ihre Projekte durchschnittlich?
   a ☐ 1 bis 25 b ☐ 26 bis 50 c ☐ 50 bis 200 d ☐ mehr als 200

8. Wie viele Komponenten werden pro Integrationsschritt durchschnittlich integriert?
   a ☐ jede Komponente einzeln c ☐ mehr als 5 gleichzeitig
   b ☐ 2 bis 5 gleichzeitig

Welcher Anteil Ihrer Projekte sind...

		100%	80%	60%	40%	20%	0%
9.	Neuentwicklungen?	☐	☐	☐	☐	☐	☐
10.	Erweiterungen durch zusätzliche Komponenten?	☐	☐	☐	☐	☐	☐
11.	Änderungen (Austausch von Komponenten)?	☐	☐	☐	☐	☐	☐

Welche Integrationsstrategien wenden Sie in Ihren Projekten an?

		nie	selten	öfter	meistens	immer
12.	Big Bang	☐	☐	☐	☐	☐
13.	Bottom up	☐	☐	☐	☐	☐
14.	Top down	☐	☐	☐	☐	☐
15.	Inside out	☐	☐	☐	☐	☐
16.	Schedule driven (Integration wird durch Zeitpunkt der Fertigstellung der Module bestimmt)	☐	☐	☐	☐	☐
17.	Hardest first (Kritische Komponenten werden zuerst integriert)	☐	☐	☐	☐	☐
18.	Feature complete (Komponenten die zusammen ein Feature bilden werden zuerst integriert)	☐	☐	☐	☐	☐
19.	_____	☐	☐	☐	☐	☐

Welche Integrationsstrategie verfolgen Sie in der Regel in folgenden Szenarien:

20. Neuentwicklungen?

21. Erweiterungen durch zusätzliche Komponenten?

22. Änderungen (Austausch von Komponenten)?

Was sind Einflussfaktoren für die Wahl der Integrationsstrategie bzw. Integrationssequenz?

		stark	**1**	**2**	**3**	**4**	**5**	schwach
23.	Expertise der Integratoren selbst		☐	☐	☐	☐	☐	
24.	von der Projektleitung vorgeben		☐	☐	☐	☐	☐	
25.	Projektzeitplan (Kundenreleases)		☐	☐	☐	☐	☐	
26.	Erfüllung von Abhängigkeiten zwischen Modulen (Vermeidung von Stubs)		☐	☐	☐	☐	☐	
27.	_____		☐	☐	☐	☐	☐	

28. Waren Sie jemals als Teil eines Integrationsteams an einem Projekt beteiligt? (Wenn nein weiter zu Frage 31)
    a ☐ ja b ☐ nein

Figure A.2: Integration Survey (print version, page 2)

Integration und Integrationstest von verteilten Software-Komponenten Seite 2

29. Wie wurde Kommunikation zwischen den Integratoren bewerkstelligt?

30. Wir wurde die Integration zwischen den einzelnen Integratoren abgestimmt?

Wie tief war Ihr Einblick in den Verlauf der Integration?

	kaum 1	2	3	4	5 vollständig
31. Verlauf der gesamten Integration war bekannt?					
32. Systemeigenschaften (Funktion und Zweck aller Module) waren bekannt?					

33. In welchem Umfang werden Stubs und Testtreiber bei der Integration eingesetzt?
kaum **1 2 3 4 5** immer

34. Sind Stubs, die für eine Komponente erstellt wurden, einer Veränderung unterworfen?
kaum **1 2 3 4 5** immer

35. Falls Stubs und Testtreiber einer Veränderung unterworfen sind, was wird verändert?

36. Werden Stubs in irgendeiner Weise verwaltet?
[a] nein [b] ja, und zwar:

Welche Probleme treten bei der Integration auf?

	kaum 1	2	3	4	5 häufig
37. Komponenten sind fehlerhaft					
38. Fehlende / falsche Dokumentation von Komponenten					
39. Inkompatible Datentypen					
40. Unterschiedliche Semantik der Schnittstellen					
41. Verletzung von Timing-Requirements					
42. Überhöhter Ressourcenbedarf					
43. Testtreiber/Stubs wurden vor der Integration nicht entfernt bzw. falsche Version wird verwendet					
44. Falsche Versionen von Komponenten wurden integriert					
45. _____					

46. Weitere häufig auftretende Probleme?

47. Was wird beim Integrationstest überprüft?

48. Wie werden Stimuli eingebracht und Ergebnisse beobachtet?

Unter welchen technischen Randbedingungen werden Integrationstest durchgeführt?

	kaum 1	2	3	4	5 immer
49. Test auf original Hardware?					
50. Emulator					
51. Hardware in the Loop?					
52. Simulation auf PC					
53. _____					

54. Gibt es einen vordefinierten Prozess falls der Integrationstest scheitert?
Wenn ja, wie sieht dieser aus?

55. Was wäre Ihrer Meinung nach nötig um den Ablauf und die Qualität der Integration und des Integrationstests zu verbessern?

Vielen Dank für Ihre Unterstützung!

Da wir unsere Forschungsergebnisse kontinuierlich hinsichtlich ihrer Anwendbarkeit in der Praxis verbessern möchten, freuen wir uns über konkrete Beispiele aus den Unternehmen.Im Gegenzug stellen wir Ihnen unsere Ergebnisse gerne zur Verfügung. Sollten Sie Interesse an einer Zusammenarbeit haben, kontaktieren Sie uns gerne!

Prof. Dr. Jürgen Mottok
Laboratory for Safe and Secure Systems
Hochschule Regensburg
www.las3.de
umfrage@las3.de

Appendix B

Action Graph Game File Example

The following presents example files in the integration planning process. The example system under integration has the following characteristics: Component 1 depends on component 0 and component 2. The timeframes for components 1 to 4 are step 5 to 10, step 5 to 8, step 2 to 5 and step 5 to 10. There are no resources defined. An intermediate file example for a system that contains four components and eleven possible integration steps looks as shows in listing B.1. The game file generator produces then the according game file (shown in listing B.2). The solution file is generated by the solver and displays the Nash equilibrium of the game (cf. listing B.3). The syntax of these files is described in detail in section 5.8.

Listing B.1: Example intermediate file.

```
p 4
s 11
d 1 0 2
t 0 5 10
t 1 5 8
t 2 2 5
t 3 5 10
```

Listing B.2: Example game file.

```
#The number of Players, n.
4
#The number of action nodes |S|.
44
#The number of function nodes, |P|.
```

```
23
#Size of action set for each Player.
11 11 11 11

#Each Player's action set.
0 4 8 12 16 20 24 28 32 36 40
1 5 9 13 17 21 25 29 33 37 41
2 6 10 14 18 22 26 30 34 38 42
3 7 11 15 19 23 27 31 35 39 43

#The Action Graph.
1 44
2 44 56
1 44
1 44
1 45
2 45 57
1 45
1 45
1 46
2 46 58
1 46
1 46
1 47
2 47 59
1 47
1 47
1 48
2 48 60
1 48
1 48
1 49
2 49 61
1 49
1 49
1 50
2 50 62
1 50
1 50
1 51
2 51 63
1 51
1 51
```

```
1 52
2 52 64
1 52
1 52
1 53
2 53 65
1 53
1 53
1 54
2 54 66
1 54
1 54
4 0 1 2 3
4 4 5 6 7
4 8 9 10 11
4 12 13 14 15
4 16 17 18 19
4 20 21 22 23
4 24 25 26 27
4 28 29 30 31
4 32 33 34 35
4 36 37 38 39
4 40 41 42 43
0
20 4 8 12 16 20 24 28 32 36 40 6 10 14 18 22 26 30 34 38 42
18 8 12 16 20 24 28 32 36 40 10 14 18 22 26 30 34 38 42
16 12 16 20 24 28 32 36 40 14 18 22 26 30 34 38 42
14 16 20 24 28 32 36 40 18 22 26 30 34 38 42
12 20 24 28 32 36 40 22 26 30 34 38 42
10 24 28 32 36 40 26 30 34 38 42
8 28 32 36 40 30 34 38 42
6 32 36 40 34 38 42
4 36 40 38 42
2 40 42
0

#Types of functions.
0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0

#The payoff functions.
0 -1.09091 -1.20202 -1.53535 -2.09091
0 -2.09091 -1.70202 -2.20202 -1.53535 -2.03535 -2.09091
0 -1.09091 -1.20202 -1.53535 -2.09091
```

0 −1.09091 −1.20202 −1.53535 −2.09091
0 −0.981818 −1.09293 −1.42626 −1.98182
0 −0.981818 −1.48182 −1.98182 −1.09293 −1.59293 −2.09293
−1.42626 −1.92626 −1.98182
0 −0.681818 −0.792929 −1.12626 −1.68182
0 −0.981818 −1.09293 −1.42626 −1.98182
0 −0.872727 −0.983838 −1.31717 −1.87273
0 −0.872727 −1.37273 −1.87273 −0.983838 −1.48384 −1.98384
−1.31717 −1.81717 −1.87273
0 −0.272727 −0.383838 −0.717172 −1.27273
0 −0.872727 −0.983838 −1.31717 −1.87273
0 −0.763636 −0.874747 −1.20808 −1.76364
0 −0.763636 −1.26364 −1.76364 −0.874747 −1.37475 −1.87475
−1.20808 −1.70808 −1.76364
0 −0.363636 −0.474747 −0.808081 −1.36364
0 −0.763636 −0.874747 −1.20808 −1.76364
0 −0.654545 −0.765657 −1.09899 −1.65455
0 −0.654545 −1.15455 −1.65455 −0.765657 −1.26566 −1.76566
−1.09899 −1.59899 −1.65455
0 −0.454545 −0.565657 −0.89899 −1.45455
0 −0.654545 −0.765657 −1.09899 −1.65455
0 −0.545455 −0.656566 −0.989899 −1.54545
0 −0.545455 −1.04545 −1.54545 −0.656566 −1.15657 −1.65657
−0.989899 −1.4899 −1.54545
0 −0.545455 −0.656566 −0.989899 −1.54545
0 −0.545455 −0.656566 −0.989899 −1.54545
0 −0.636364 −0.747475 −1.08081 −1.63636
0 −0.636364 −1.13636 −1.63636 −0.747475 −1.24747 −1.74747
−1.08081 −1.58081 −1.63636
0 −0.80303 −0.914141 −1.24747 −1.80303
0 −0.636364 −0.747475 −1.08081 −1.63636
0 −0.727273 −0.838384 −1.17172 −1.72727
0 −0.727273 −1.22727 −1.72727 −0.838384 −1.33838 −1.83838
−1.17172 −1.67172 −1.72727
0 −1.06061 −1.17172 −1.50505 −2.06061
0 −0.727273 −0.838384 −1.17172 −1.72727
0 −0.818182 −0.929293 −1.26263 −1.81818
0 −0.818182 −1.31818 −1.81818 −0.929293 −1.42929 −1.92929
−1.26263 −1.76263 −1.81818
0 −1.31818 −1.42929 −1.76263 −2.31818
0 −0.818182 −0.929293 −1.26263 −1.81818
0 −0.909091 −1.0202 −1.35354 −1.90909
0 −1.24242 −1.74242 −2.24242 −1.35354 −1.85354 −2.35354

```
  −1.68687 −2.18687 −2.24242
0 −1.57576 −1.68687 −2.0202 −2.57576
0 −0.909091 −1.0202 −1.35354 −1.90909
0 −1 −1.11111 −1.44444 −2
0 −1.66667 −1.77778 −2.11111 −2.66667
0 −1.83333 −1.94444 −2.27778 −2.83333
0 −1 −1.11111 −1.44444 −2
```

Listing B.3: Solution file.

```
testing GW algorithm:
run #0
     0      0      0      0      0      0      1      0
0      0      0      0      0      0      0      0      0
0      1      0      0      0      0      0      1      0
0      0      0      0      0      0      0      0      0
0      0      0      1      0      0      0      0      0
Expected utility for each player under a sample eqlm:
−0.636364 −0.727273 −0.272727 −0.545455
Expected configuration of a sample eqlm:
     0      0      0      0      0      0      0      0
0      0      1      0      0      0      0      0      0
0      0      0      0      0      0      1      1      0
0      0      0      1      0      0      0      0      0
0      0      0      0      0      0      0      0      0
CPU Time for AGG 2.02

Avg CPU Time :
2.02
```

Bibliography

Adler, B. T., Alfaro, L. D., Silva, R. D. D., Faella, M., Legay, A., Raman, V., & Roy, P. (2006). Ticc, a tool for interface compatibility and composition. In *In Proceedings 18th International Conference on Computer Aided Verification (CAV), volume 4144 of Lecture Notes in Computer Science* (pp. 59–62). Springer.

Alter, S. (1975). *A study of computer aided decision making in organizations* (PhD thesis, Massachusetts Institute of Technology.).

Artop User Group. (2012). Autosar Tool Platform User Group. https://www.artop.org/.

ATESST2. (2012). EAST-ADL2 specs, profile and tools. http://www.atesst.org/.

ATESST2 & KTH. (2011). A connection between Simulink, Eclipse Modeling Framework (EMF) and EAST-ADL. http://code.google.com/p/kth-simulink-exchange/.

AUTOSAR Administration. (2008). *Technical Overview R3.1*. AUTOSAR GbR.

Avriel, M. (2003). *Nonlinear Programming: Analysis and Methods*. Dover Publications.

Balzert, H. (2008). *Lehrbuch der Softwaretechnik: Softwaremanagement*. Spektrum Akademischer Verlag.

Bargiacchi, D., Jiang, A. X., & Leyton-Brown, K. (2012). AGGUI. http://agg.cs.ubc.ca/.

Baumgartner, M., Sneed, H. M., & Seidl, R. (2010 2013/08/03). Software in zahlen. In *Software in Zahlen* (pp. I–XIX). Carl Hanser Verlag GmbH & Co. KG. doi:doi:10.3139/9783446424487.fm

Beizer, B. (1984). *Software system testing and quality assurance*.

Bertolino, A., Inverardi, P., & Muccini, H. (2003). Formal methods in testing software architectures. *Formal methods for software architectures*, 122–147.

Bertolino, A., Inverardi, P., Muccini, H., & Rosetti, A. (1997). An approach to integration testing based on architectural descriptions. In *ICECCS* (p. 77). Published by the IEEE Computer Society.

Beydeda, S. & Gruhn, V. (2003). Merging components and testing tools: The self-testing COTS components (STECC) strategy.

Bhat, N. A. R. & Leyton-Brown, K. (2004). Computing nash equilibria of action-graph games. In *Proceedings of the 20th conference on uncertainty in artificial intelligence* (pp. 35–42). UAI '04. Banff, Canada: AUAI Press. **retrieved from** http://dl.acm.org/citation.cfm?id=1036843.1036848

Binder, R. V. (1994 Sept.). Design for testability in object-oriented systems. *Commun. ACM*, *37*, 87–101. doi:10.1145/182987.184077

Binder, R. V. (1999). *Testing object-oriented systems: models, patterns, and tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

BITKOM. (2010). Eingebettete Systeme - Ein strategisches Wachstumsfeld für Deutschland: Anwendungsbeispiele, Zahlen und Trends.

Boehm, B. (1979). Guidelines for Verifying and Validating Software Requirements and Design Specifications. In *In Proceedings of Euro IFIP 1979* (pp. 711–719). eprint: http://csse.usc.edu/csse/TECHRPTS/1979/usccse79-501/usccse79-501.pdf

Borner, L. & Paech, B. (2009). Integration Test Order Strategies to Consider Test Focus and Simulation Effort. In *Advances in system testing and validation lifecycle, 2009. valid'09. first international conference on* (pp. 80–85). IEEE.

Bourque, P. & Dupuis, R. (2004). Guide to the software engineering body of knowledge 2004 version. *Software Engineering Body of Knowledge 2004 SWEBOK, Guide to the*.

Briand, L. C., Labiche, Y., & Wang, Y. (2003 July). An investigation of graph-based class integration test order strategies. *IEEE Trans. Softw. Eng.* *29*(7), 594–607. doi:10.1109/TSE.2003.1214324

Brwan, A. (1997). Background Information on CBD. SIGPC.

Burstein, F. & Holsapple, C. (2008). *Handbook on Decision Support Systems 1*. International Handbooks on Information Systems. Springer Berlin Heidelberg.

Cachon, G. P. & Netessine, S. (2006). Game theory in supply chain analysis. *Tutorials in Operations Research: Models, Methods, and Applications for Innovative Decision Making*.

CEA LIST. (2012). Papyrus for EAST-ADL. http://www.papyrusuml.org/scripts/home/publigen/content/templates/show.asp?P=146&L=EN&ITEMID=14.

Chidamber, S. & Kemerer, C. (1994 June). A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, *20*(6), 476–493. doi:10.1109/32.295895

Claraz, D., Eppinger, K., & Berentroth, L. (2004a). Reuse strategy at siemens VDO automotive the EMS 2 powertrain platform architecture. *Ingenieurs de l Automobile*, 767.

Claraz, D., Eppinger, K., & Berentroth, L. (2004b). Reuse strategy at siemens VDO automotive: the EMS 2 powertrain platform architecture. *Ingenieurs de l'Automobile*, *767*.

Clarke, J. (1998). Automated test generation from a behavioral model. In *Proceedings of Pacific Northwest Software Quality Conference. IEEE Press*. Citeseer.

Crnkovic, I. (2005). Component-based software engineering for embedded systems. In *Proceedings of the 27th international conference on Software engineering* (pp. 712–713). ACM.

Crnkovic, I. (2001). Component-based software engineering: new challenges in software development. *Software Focus*, *2*(4), 127–133.

Cuenot, P., Chen, D., Gerard, S., Lönn, H., Reiser, M.-O., Servat, D., ... Weber, M. (2007). Managing Complexity of Automotive Electronics Using the EAST-ADL. In *ICECCS '07: Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems* (pp. 353–358). Washington, DC, USA: IEEE Computer Society. doi:10.1109/ICECCS.2007.28

de Alfaro, L. & Henzinger, T. A. (2001a). Interface automata. *SIGSOFT Software Engineering Notes*, *26*(5), 109–120. doi:10.1145/503271.503226

de Alfaro, L. & Henzinger, T. A. (2001b). Interface Theories for Component-Based Design. In *EMSOFT '01: Proceedings of the First International Workshop on Embedded Software* (pp. 148–165). London, UK: Springer-Verlag.

de Alfaro, L. & Henzinger, T. A. (2005). Interface-based Design. In *Engineering Theories of Software-intensive Systems* (Vol. 195, pp. 83–104). NATO Science Series: Mathematics, Physics, and Chemistry. Springer. M. Broy, J. Gruenbauer, D. Harel, and C.A.R. Hoare.

de Alfaro, L., Henzinger, T. A., & Stoelinga, M. (2002). Timed Interfaces. In *EMSOFT* (Vol. 2491, pp. 108–122). Lecture Notes in Computer Science. Springer.

de Alfaro, L. & Stoelinga, M. (2004). Interfaces: A Game-Theoretic Framework for Reasoning About Component-Based Systems. *Electr. Notes Theoretical Computer Science*, *97*, 3–23.

Dorigo, M. (1992). *Optimization, Learning and Natural Algorithms (in Italian)* (PhD thesis, Dipartimento di Elettronica, Politecnico di Milano, Milan, Italy).

Eclipse Foundation. (2012a). Eclipse M2M Project. http://www.eclipse.org/m2m/.

Eclipse Foundation. (2012b). Eclipse Modeling Project. http://www.eclipse.org/modeling/emf/.

Edelkamp, S. & Schrödl, S. (2012). *Heuristic Search - Theory and Applications*. Academic Press.

Edwards, S. (2001). A framework for practical, automated black-box testing of component-based software. *Software Testing, Verification and Reliability*, *11*(2), 97–111.

Eickelmann, N. & Richardson, D. (1996). What makes one software architecture more testable than another? In *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints' 96) on SIGSOFT'96 workshops* (pp. 65–67). ACM.

El-Far, I. & Whittaker, J. (2001). Model-based software testing. *Encyclopedia of Software Engineering*.

Feghali, I., Watson, A., Henderson-Sellers, B., & Tegarden, D. (1994). Technical Correcspondence: Clarrification concerning modularization and McCabe's cyclomatic complexity. *Communications of the ACM*, *37*(4), 91–94.

Gao, J., Tsao, H., & Wu, Y. (2003). *Testing and quality assurance for component-based software*. Artech House on Demand.

Gąsior, D. & Drwal, M. (2012). Pareto-optimal Nash equilibrium in capacity allocation game for self-managed networks. *arXiv preprint arXiv:1206.2448*.

Giusto, P., Ferrari, A., Lavagno, L., Brunel, J.-Y., Fourgeau, E., & Sangiovanni-Vincentelli, A. (2002). Automotive virtual integration platforms: why's, what's, and how's. In *Computer Design: VLSI in Computers and Processors, 2002. Proceedings. 2002 IEEE International Conference on* (pp. 370–378). doi:10.1109/ICCD.2002.1106796

Govindan, S. & Wilson, R. (2003). A global Newton method to compute Nash equilibria. *Journal of Economic Theory*, *110*(1), 65–86.

Hafner, M. (2010). Umfrage "Software-Integration". *ELEKTRONIKPRAXIS - ESE-Report April 2010*. **retrieved** 22. July 2013, **from** http://files.vogel.de/vogelonline/vogelonline/issues/ep/2010/2629.pdf

Harsanyi, J. C. (2004). Games with Incomplete Information Played by Bayesian Players, I–III: Part I. The Basic Model. *Management science*, *50*(12 supplement), 1804–1817.

Hartmann, J., Imoberdorf, C., & Meisinger, M. (2000). UML-based integration testing. In *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis* (pp. 60–70). ACM.

Hiemann, P. (1975). A new look at the program development process. *Programming Methodology*, 11–37.

Holland, J. H. (1975). *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence*. U Michigan Press.

Holler, M. J. & Illing, G. (2006). *Einführung in die Spieltheorie*. Springer.

Holsapple, C., Whinston, A., Benamati, J., & Kearns, G. (1996). *Decision support systems: a knowledge-based approach*. West Pub. Co.

IBM, IDC, Mercer, & M. Analysis. (2010). ARC Advisory: Global Industry Analysts.

Jay, F. & Mayer, R. (1990). IEEE standard glossary of software engineering terminology. *IEEE Std*, *610*, 1990.

Jersak, M. (October 2007). Timing model and methodology for AUTOSAR. *Elektronik Automotive*.

Jiang, A. X. (2011). Agg/bagg solver. http://agg.cs.ubc.ca/.

Jiang, A. X. & Leyton-Brown, K. (2010). Bayesian action-graph games. *Advances in Neural Information Processing Systems*, *23*, 991–999.

Jiang, A. X., Leyton-Brown, K., & Bhat, N. (2010). Action-graph games. *Games and Economic Behavior*.

Jung, M. & Saglietti, F. (2005). Supporting Component and Architectural Re-usage by Detection and Tolerance of Integration Faults.

Kanajan, S., Zeng, H., Pinello, C., & Sangiovanni-Vincentelli, A. (2006). Exploring trade-off's between centralized versus decentralized automotive architectures using a virtual integration environment. In *Proceedings of the conference on Design, automation and test in Europe: Proceedings* (pp. 548–553). DATE '06. Munich, Germany: European Design and Automation Association. **retrieved from** http://dl.acm.org/citation.cfm?id=1131481.1131632

Kearns, M., Littman, M., & Singh, S. (2001). Graphical models for game theory. In *Proceedings of the 17th conference in uncertainty in artificial intelligence* (pp. 253–260).

Keen, P. & Morton, M. (1978). *Decision support systems: an organizational perspective*. Addison-Wesley Pub. Co.

Kennedy, J. & Eberhart, R. (1995). Particle Swarm Optimization. In *Neural networks, 1995. proceedings., ieee international conference on* (Vol. 4, 1942–1948 vol.4). doi:10.1109/ICNN.1995.488968

Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, *220*, 671–680.

Kopetz, H. (1998). Component-based design of large distributed real-time systems. *Control Engineering Practice*, *6*(1), 53–60.

Koutsoupias, E. & Papadimitriou, C. (1999). Worst-case equilibria. In *Proceedings of the 16th annual conference on Theoretical aspects of computer science* (pp. 404–413). Springer-Verlag.

Kuenzli, S. (2006). *Efficient design space exploration for embedded systems* (PhD thesis, Swiss Federal Institute of Technology Zurich, Switzerland).

Kuhlen, R., Seeger, T., & Strauch, D. (2004). *Grundlagen der praktischen Information und Dokumentation*. KG Saur.

Kung, D., Gao, J., HsiaYasufumi, P., & Chen, C. (1996). On regression testing of object-oriented programs. *Journal of Systems and Software*, *32*(1), 21–40.

Kunz, W. & Rittel, H. (1972). *Die Informationswissenschaften: Ihre Ansaetze, Probleme, Methoden und ihr Ausbau in der Bundesrepublik Deutschland*. Oldenbourg.

Laboratory for Safe and Secure Systems. (2009). Vitas[3] homepage. **retrieved from** http://www.las3.de/index.php?id=28

Le Traon, Y., Jéron, T., Jézéquel, J., & Morel, P. (2002). Efficient object-oriented integration and regression testing. *Reliability, IEEE Transactions on*, *49*(1), 12–25.

Liggesmeyer, P. (2009). *Software-Qualiät: Testen, Analysieren und Verifizieren von Software*. Springer.

Linnenkugel, U. & Mullerburg, M. (1990). Test data selection criteria for (software) integration testing. In *Proceedings of the First International Conference on Systems Integration, 1990. Systems Integration'90.* (Pp. 709–717).

Lönn, H., Berntsson, L.-O., Blom, H., Chen, D., Cuenot, P., Donandt, J., ... Weber, M. (2008). *EAST ADL 2.0 Specification*. ATESST Consortium.

Luce, R. D. & Raiffa, H. (1957). *Games and decisions: Introduction and critical survey*. Courier Dover Publications.

Ma, L., Wang, H., & Lu, Y. (2006). The Design of Dependency Relationships Matrix to improve the testability of Component-based Software. In *QSIC '06: Proceedings of the Sixth International Conference on Quality Software* (pp. 93–98). Washington, DC, USA: IEEE Computer Society. doi:http://dx.doi.org/10.1109/QSIC.2006.64

MathWorks, Inc. (2011). Automobilindustrie - Ubersichtsseite. **retrieved from** http://www.mathworks.de/automotive/

MathWorks, Inc. (2000). *Using Simulink and Stateflow in Automotive Applications*.

McGregor, J. & Sykes, D. (2001). *A practical guide to testing object-oriented software*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.

Monderer, D. & Shapley, L. S. (1996). Potential games. *Games and Economic Behavior*, *14*(1), 124–143. doi:10.1006/game.1996.0044

Mottok, J. (2009). Bayerisches IT-Sicherheitscluster: Automotive Forum. **retrieved from** http://www.it-speicher.de/itsecurity

Mottok, J., Kuntz, S., & Niemetz, M. (2008). *Vorhabensbeschreibung für das Förderprogramm "IngenieurNachwuchs"2008 (Elektrotechnik) - VitaS[3] "Virtuelle und Automatisierte Integration von Softwarefunktionen in verteilten eingebetteten Automobil-Systemen unter Berücksichtigung der Anforderungen an die funktionale Sicherheit"*.

Object Management Group. (2012). Meta Object Facility. http://www.omg.org/mof/.

Object Management Group. (2011). *Meta object facility (mof) 2.0 query/view/transformation specification*. Object Management Group.

Object Management Group. (2007 Nov.). *OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2*. eprint: \url{http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF}

Panyr, J. (1986). *Automatische Klassifikation und Information Retrieval*. Niemeyer.

Paul, R. (2001). End-to-end Integration Testing. In *APAQS*. Published by the IEEE Computer Society.

Pohl, K. (2008). *Requirements Engineering - Grundlagen, Prinzipien, Techniken (2. Aufl.)* dpunkt.verlag.

Project, M. (2011). MPXJ Project Website. http://mpxj.sourceforge.net/.

Rehman, M., Jabeen, F., Bertolino, A., & Polini, A. (2007). Testing software components for integration: a survey of issues and techniques. *Software Testing, Verification and Reliability*, *17*(2), 95–133.

Rosaria, S. & Robinson, H. (2000). Applying models in your testing process. *Information and Software Technology*, *42*(12), 815–824.

Rowley, J. (2008). *Organizing knowledge: an introduction to managing access to information*. Ashgate Pub Co.

Rudorfer, M., Voget, S., & Eberle, S. (2010). *Artop Whitepaper*.

Rumbaugh, J., Jacobson, I., & Booch, G. (2004). *Unified Modeling Language Reference Manual, The*. Pearson Higher Education.

Saglietti, F., Oster, N., & Pinte, F. (2008). Interface Coverage Criteria Supporting Model-Based Integration Testing. *ARCS'07*.

Sandmann, G. & Schlosser, J. (2008). Entwicklung von AUTOSAR Software-Komponenten mit Model-Based Design. http://www.elektroniknet.de/automotive/technik-know-how/test-entwicklungstools/article/1168/1/Entwicklung_von_AUTOSAR_Software-Komponenten_mit_Model-Based_Design/.

Schäuffele, J. & Zurawka, T. (2010). *Automotive Software Engineering*. Vieweg + Teubner, Wiesbaden.

Schorer, M. (2011a). *Integration Cost Analysis at Continental Automotive GmbH*. Technical Report, Project VitaS^3, Laboratory for Safe and Secure Systems (LaS^3), OTH Regensburg.

Schorer, M. (2011b). *Requirements for an Integration Support Toolchain*. Technical Report, Project VitaS^3, Laboratory for Safe and Secure Systems (LaS^3), OTH Regensburg.

Schorer, M. (2010a). *State of the Art Report - Software Integration at Continental Automotive GmbH*. Technical Report, Project VitaS^3, Laboratory for Safe and Secure Systems (LaS^3), OTH Regensburg.

Schorer, M. (2010b). *State of the Art Report - Workshop on Software Integration at iNTENCE Automotive GmbH*. Technical Report, Project VitaS^3, Laboratory for Safe and Secure Systems (LaS^3), OTH Regensburg.

Schucan, C. (1999). *Effektivitätssteigerung mittels konzeptionellem Informationsmanagement* (PhD thesis).

Scott Morton, M. (1971). *Management decision systems: computer-based support for decision making*. Division of Research, Graduate School of Business Administration, Harvard University (Boston).

SEI. (2012). Glossary of Software Architecture Terms: Software Architecture. http://www.sei.cmu.edu/architecture/index.cfm.

Sim, K.-B., Lee, D.-W., & Kim, J.-Y. (2004). Game theory based coevolutionary algorithm: a new computational coevolutionary approach. *International Journal on Control Automation Systems*, *2*(4), 463–474.

Singh, G., Singh, D., & Singh, V. (2011). A Study of Software metrics. *International Journal of Computational Engineering and Management*, *11*, 2230–7893.

Spillner, A. & Linz, T. (2012). *Basiswissen Softwaretest*. dpunkt.verlag.

Stappert, F., Sjöstedt, C. J., Abele, A., Hagl, F., Lönn, H., & Papadopoulos, Y. (2010). *Case study and demonstrator plan*. ATESST Consortium.

Steinberg, D. (2008). Fundamentals of the eclipse modeling framework. http://www.eclipse.org/modeling/emf/docs/presentations/EclipseCon/EclipseCon2008_309T_Fundamentals_of_EMF.pdf.

Szyperski, C., Gruntz, D., & Murer, S. (2002). Component software: beyond object-oriented programming. Addison-Wesley.

Tai, K. & Daniels, F. (2002). Test order for inter-class integration testing of object-oriented software. In *Computer software and applications conference, 1997. compsac'97. proceedings., the twenty-first annual international* (pp. 602–607). IEEE.

TUM. (2012). ConQAT. http://conqat.in.tum.de/index.php/Simulink_Library.

van Der Laan, G., Talman, A., & Van der Heyden, L. (1987). Simplicial variable dimension algorithms for solving the nonlinear complementarity problem on a product of unit simplices using a general labelling. *Mathematics of operations research*, 377–397.

Wang, Y., King, G., & Wickburg, H. (1999). A method for built-in tests in component-based software maintenance. In *Proceedings of the Third European Conference on Software Maintenance and Reengineering, 1999* (pp. 186–189).

Watson, A. (1996). Structured testing: Analysis and extensions. *Princeton University, Princeton, NJ*.

Watson, A., McCabe, T., & Wallace, D. (1996). Structured testing: A testing methodology using the cyclomatic complexity metric. *NIST Special Publication*, *500*, 235.

Wu, Y., Chen, M., & Offutt, J. (2003). UML-based integration testing for component-based software. *COTS-Based Software Systems*, 251–260.

Wu, Y. & Dai Pan, M. (2001). Techniques for testing component-based software. In *Iceccs* (p. 0222). Published by the IEEE Computer Society.

Zehnder, C. A. (2005). *Informationssysteme und Datenbanken (8. Aufl.)* Vdf Hochschulverlag.

Zhu, H., Hall, P., & May, J. (1997). Software unit test coverage and adequacy. *ACM Computing Surveys (CSUR)*, *29*(4), 427.

Zhu, H. & He, X. (2001). An observational theory of integration testing for component-based software development. In *Computer Software and Applications Conference, 2001. COMPSAC 2001. 25th Annual International* (pp. 363–368).